

n

Copyright © 2017 by Michael Galea
All Rights Reserved

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Monday, April 27, 2015
2:04 PM

PowerShell ebooks

<http://powershell.org/wp/ebooks/>

<http://it-ebooks.info/tag/powershell/>

<https://technet.microsoft.com/en-us/library/bb978526.aspx>

Video" "Learn Power Shell in a Month of Lunches"

https://www.youtube.com/results?search_query=Learn+Windows+PowerShell+in+a+Month+of+Lunches%22

Introduction

Saturday, December 21, 2013
8:18 AM

http://en.wikipedia.org/wiki/Windows_PowerShell

PowerShell was specifically designed for Windows administration. However, it has become a fully featured scripting language in the tradition of Unix/Linux shells. Unlike the nc shells where commands receive and output text, every datum is PowerShell is an *object*. An object is a data structure that contains properties that describe the object and behaviors, called methods, that impact the object. Another important difference that makes PowerShell unique is the interface to .Net, WMI, and COM which extends the functionality of PowerShell.

On August 2016, PowerShell was open-sourced by Microsoft making it available sometime after that date on Linux and Mac platforms.

PowerShell in some respects is a replacement for the cmd (command) shell. In fact, on Windows Server 2012 and beyond running in core mode, it is possible to replace the cmd shell with Windows PowerShell so that when the server boots up, it uses PowerShell as the interface.

For legacy support, PowerShell also supports the cmd external commands. However, in many cases, using DOS external commands like ping or xcopy isn't necessary since PowerShell has more than adequate replacements.

PowerShell commands are called cmdlets. PowerShell is case insensitive so that cmdlets and statements may be written in any case. A cmdlet name begins with an action verb: *set, get, put, copy, ...* and usually describes the specific function that it performs, e.g., copy-item copies a file or directory from one location to another. PowerShell cmdlets are also *polymorphic*. This means, that unlike the cmd shell, different cmdlets are not required to execute the same action against

different objects. For example the cmdlet copy-item mentioned above not only works against files and directories but also against registry keys, and certificate objects.

Similar to other scripting languages, PowerShell supports:

Functions	A series of PowerShell cmdlets and scripting statements that return or produce a result. A function exists once within a script and may be called multiple times. It may be standalone and called from the command line , a script, or another function.
Aliases	Aliases are useful in reducing the verbosity of PowerShell. For example, get-childitem has an alias gci. Aliases are also used in the transition from the cmd shell to PowerShell. For example, the cmdlet get-childitem has an alias dir and ls.
Modules	Modules extend the functionality of PowerShell by adding cmdlets, providers, functions, variables, aliases, and much more

A script in PowerShell is a text file with a .ps1 extension that may contain:

- A series of cmdlets.
- A combination of PowerShell scripting statements with cmdlets.
- All of the above plus objects created from .Net classes.

The term scriptblock is used throughout this document. A scriptblock is just a series of PowerShell statements and/or cmdlets delimited by braces. Scriptblocks are used by certain cmdlets like where-object and foreach-object and in a number of PowerShell control statements.

PowerShell exists in two forms, the PowerShell Console and the PowerShell Integrated Scripting Environment (ISE). The executables for both the console and the ISE are found in path "c:\Windows\System32\WindowsPowerShell\v1.0". When executed, the Console appears very similar to the legacy command (DOS) shell with the one difference is that the prompt begins with "PS", and like the cmd shell, followed by the path of the current location on the file system.



The ISE is used from script development. More information on the ISE will be presented later.

Writing scripts is essentially programming. The good news is that you don't need to know a lot about programming to create PowerShell scripts. However, since using scripting statements makes for more powerful scripts, we will learn about writing scripting code.

Intro to cmdlets

Tuesday, June 17, 2014
2:36 PM

Cmdlets is just another name for commands executed in the PowerShell console. Cmdlets are programs written in C# that provide functionality within the OS. Hundreds of cmdlets are available and more become available with each release.

To better understand cmdlets, below are some common cmdlets and their aliases. Let's look at some common cmdlets and their aliases using some of the commonly used cmd shell commands. In the examples below lines beginning with # are comments.

Below the first item is the legacy cmd (DOS) shell command, followed by the equivalent PowerShell cmdlet which is followed by the corresponding aliases. As you will notice, some of the aliases names are the same as the equivalent command in the DOS or Linux shell. However, only the name is the same. The parameters and switches required to execute that command conform the PowerShell cmdlet syntax and not to the DOS or Linux syntax.

dir **Get-Childitem** gci, dir, and ls

Get-Childitem is the functional equivalent of dir with the exception that select and inclusion providers are available to select by file system object type or select file system objects based upon wildcards.

```
# List the files and subdirectories in the current directory.  
Get-ChildItem  
  
# List the files and subdirectories in the current directory and all subdirectories.  
Get-Childitem -Recurse  
  
# List only system files in the Windows directory and its subdirectories.  
Get-Childitem -path c:\windows -System -File -Recurse  
  
# List all files in the current directory ,i.e., exclude directories.  
gci -Attributes !directory # use the alias gci  
ls -att !d  
dir -file  
gci -file  
  
# List only directories in the current directory.  
gci -attributes d  
dir -att d  
ls -directory  
Get-Childitem -directory  
  
# List hidden files  
gci -file -attributes hidden  
  
# List all read-only files in the directory users and its subdirectories.  
gci -file -attributes readonly -path c:\users -recurse  
gci c:\users -file -attributes readonly -recurse  
  
# List all read-only text files in the users directory and its subdirectories  
gci -file -att r -path c:\users -include *.txt -recurse
```

cd **Set-Location** sl and cd

Set-location functions differently than cd. The cd command is used to change the location on the system for the current drive or another local drive. Set-Location functions identically to cd when operating on the same volume as the current location, in that, it changes the location or attaches to the specified path. When specifying other locations, this cmdlet moves the location of the PS session to the specified location and attaches to the specified path. As shown below, the location may be a virtualized PowerShell location like the Registry, the Environment Variables store, the Certificate store, remote host location., and others.

```
# Set the location to c:\windows
Set-Location c:\windows

# Set the location to the parent
Set-Location ..

# Set the location to the subdirectory data in the d: volume
sl d:\data

# Set the location to user profile directory
sl ~

# Set the location to environment drive (environment variables)
sl env:

# Set the location to the HKEY_LOCAL_MACHINE in the Registry
sl hklm:

# Set the location to the g$ administrative share on the host server1.
set-location \\server1\g\$
```

del Remove-Item ri, del, erase, rd, rm, and rmdir

As is implied by the aliases, Remove-Item combines the functionality of the CMD commands ri, del, erase, rd, rm, and rmdir. The aliases are:

```
# remove the specified file
Remove-Item e:\files\readme.txt

# remove the directory and all subdirectories
Remove-Item e:\files -recurse

# Remove all text files in the specified directory and sub-directory
# except any text file whose name begins with sp
```

```
Remove-Item e:\files -include *.txt -exclude sp*.txt -recurse  
# same as the above command but will delete read-only and hidden text files  
Remove-Item e:\files -include *.txt -exclude sp*.txt -recurse -force
```

md New-Item ni

Unlike the CMD command `md`, `New-Item` can create directories and files on the files system, values in the Registry, and other objects on virtualized drives.

```
# create the directory files in the root of e:  
New-Item e:\files -ItemType directory  
  
# create the text file test.txt in e:\files  
new-item e:\files\test.txt -itemtype file  
new-item -path e:\files -name test.txt -itemtype file  
  
# same as above but populate file with two lines of text  
new-item -path e:\files -name test.txt -itemtype file `  
-value "This is line 1.`nThis is line 2"
```

In the above, the second example the two cmdlets both create an empty file called `test.txt` in the directory `e:\files`.

The third example differs from the second in that the file is populated with the string specified by `-value`. The "`n`" is called a metacharacter and designates the new-line character. So the sentence preceding the new-line character is the first line of the text file. The sentence succeeding the new-line character is the second line of the text file. Note also the grave, or back-tick, character following the first line. This is a line continuation character which causes PowerShell continue looking for input on the following line.

Help

Friday, August 2, 2013
4:48 PM

An extensive help system is available with PowerShell.

```

# list available cmdlets, functions, and aliases
get-command

# list available aliases
get-command -commandtype alias

# Downloads and installs the latest help files.
# Use this if you can't find help for a cmdlet.
# This requires a console running as Administrator
update-help

# get-help for the cmdlet Get-childitem
get-help get-childitem

# This accomplished the same as the above
# help is an alias for get-help.  gci is an alias for get-childitem
help gci

# The-full switch results in a more detailed help
help get-childitem -full

# The -examples switch displays a number of usage examples
get-help get-childitem -examples

# Wildcards may be used
help get-child*

# Show all cmdlets that have service in the name.
help *service*

# help on help is also available
get-help get-help

```

Help parameters:

-Full	displays the full help, including details for each command parameter and usually including usage examples.
-Examples	Displays usage examples only.
-Detailed	Displays details on each command parameter but doesn't display usage examples.
-Online	Opens the help in the system's default web browser, loading from Microsoft's website the most current help.

-ShowWindow	Opens full help in a pop-up window.
-------------	-------------------------------------

Interpreting help:

When getting help a particular cmdlet, multiple syntax diagrams may be displayed. Each syntax diagram represents the unique parameter set that may be used with the cmdlet.

```
PS C:\Users>
PS C:\Users> help Get-ChildItem

NAME
    Get-ChildItem

SYNOPSIS
    Gets the files and folders in a file system drive.

SYNTAX
    Get-ChildItem [[-Path] <String[]>] [[-Filter] <String>] [-Exclude <String[]>]
    [-UseTransaction [<SwitchParameter>]] [<CommonParameters>]

    Get-ChildItem [[-Filter] <String>] [-Exclude <String[]>] [-Force [<SwitchParameter>]]
    [-UseTransaction [<SwitchParameter>]] [<CommonParameters>]

    Get-ChildItem [-Attributes <FileAttributes>] [-Directory] [-File] [-Force]

DESCRIPTION
    The Get-ChildItem cmdlet gets the items in one or more specified locations.
    By default, Get-ChildItem gets non-hidden items, but you can use the -Hidden parameter to get items in all child containers.
    A location can be a file system location, such as a directory, or a location in a provider. In a file system drive, the Get-ChildItem cmdlet gets the directories, subdirectories, and files at that location.
    By default, Get-ChildItem gets non-hidden items, but you can use the -Hidden parameter to get hidden items. To search for items based on their attributes, use the Attributes parameter. If you use these parameters together, use the -And operator.
    To see the examples, type: "get-help Get-ChildItem -examples".
    For more information, type: "get-help Get-ChildItem -detailed".
    For technical information, type: "get-help Get-ChildItem -full".
    For online help, type: "get-help Get-ChildItem -online"

RELATED LINKS
    http://go.microsoft.com/fwlink/?LinkId=204557
    about_Providers
    FileSystem Provider
    Get-Item

REMARKS
```

Note that the third syntax diagram has the parameter `-Attributes` which is not include in the other two. This syntax diagram also does not include the `-Name` parameter which is include in the other two. This means that the `-Attributes` and `-Name` parameters are mutually exclusive and may not be used simultaneously when crafting the cmdlet usage syntax.

In the above, the brackets [] indicate an optional item. For example, the `get-childitem` cmdlet may be executed as below since all parameters are optional:

```
get-childitem
```

It may also be executed using the syntax below:

```
get-childitem -path c:\windows
```

The syntax `[[-Path] <string>]` implies that `-Path` is optional. So, the syntax below is also acceptable:

```
get-childitem c:\windows
```

Parameters that do not have a value following them are called *switch parameters*. Below `-force` and `-recurse` are switch parameters.

```
get-childitem c:\windows -recurse -force
```

About help files:

PowerShell includes help for general concepts, troubleshooting, and so forth usually referred to as “about” files because their filenames start with the word “about”.

About help acts as the formal PowerShell documentation.

```
help about*
```

```
help about_aliases
```

Common Parameters are parameters common to all cmdlets. Information about the common parameters is available through:

```
get-help about_common*
```

If you need to better understand the syntax diagrams used in help for cmdlets execute the command below.

```
get-help about_command_syntax
```

Common Parameters

Friday, June 13, 2014
8:48 PM

Cmdlets have a set of common parameters which are implemented to provide some useful and commonly used features. Common Parameters include two risk mitigation parameters:

-WhatIf and -Confirm.

Many times the user would like to know what the effects of a cmdlet without actually executing the cmdlet. For example,

```
remove-item * -recurse
```

The above syntax would remove all items in the current directory and all subdirectories contained within the current directory. The common parameter *-whatif* allows the user to understand all the files and directories would be removed without actually removing them.

```
remove-item * -recurse -whatif
```

The *-confirm* common parameter requires the user to confirm every step of the command.

```
remove-item * -recurse -confirm
```

In the above example, the user would be required to confirm every file to be deleted.

Other Common Parameters are listed in the table below.

-Verbose	Switch	Generates as much information as possible. Without this switch, the cmdlet restricts itself to displaying only essential information
-Debug	Switch	Outputs additional warnings and error messages that help programmers find the causes of errors.
-ErrorAction	Value	Determines how the cmdlet responds when an error occurs. Permitted values: <i>Continue</i> : reports error and continues (default)

		<p><i>Stop</i>: reports error and stops <i>SilentContinue</i>: displays no error message, continues <i>SilentStop</i>: displays no error message, stops <i>Inquire</i>: asks how to proceed</p>
-ErrorVariable	Value	Name of a variable in which in the event of an error information about the error is stored.
-OutVariable	Value	<p>Name of a variable in which the result of a cmdlet is to be stored. This parameter is usually superfluous because you can directly assign the value to a variable. The difference is that it will no longer be displayed in the console if you assign the result to a variable.</p> <p><code>\$result = Get-ChildItem</code></p> <p>It will be output to the console and stored in a variable if you assign the result additionally to a variable:</p> <p><code>Get-ChildItem -OutVariable result</code></p>
-WhatIf	Switch	Describes the effect of the cmdlet instead of executing it.
-Confirm	Switch	Prompts the user for confirmation before executing the command

More information about Common Parameters is available by executing help as below.

```
help about_common_parameters
```

Variables

Thursday, December 21, 2017
3:32 PM

Similar to all other scripting languages, PowerShell supports the use of variables. Variables are used to allocate space within the address space of the PowerShell session that is used to store data for later processing.

```
PS> 20
20
PS> $i = 20
PS>
PS> $i
20
PS> write-host $i
20
PS> "The number is $i"
The number is 20
PS>
PS> write-host "The number is $i"
The number is 20
PS>
PS>
```

In the statement sequence shown above, the following is happening:

1. 20 is entered in the console causing PowerShell to display the value back on the console.
2. The value 20 is then assigned to the variable \$i. Note that all PowerShell variable names must be preceded by "\$". Note also that the character "=" is the assignment operator and does not mean equal. The second statement would be read "The variable \$i is assigned the value 20".
3. \$i is entered in the console causing PowerShell to display the contents of the variable.
4. The write-host cmdlet is used to display the contents of the variable \$i on the console.
5. A double-quoted string is entered with the \$i variable embedded causing PowerShell to display the string with the variable expanded.
6. The write-host cmdlet is used to display the string.

Variables are also typed based upon the type of object they hold. The `GetType` method provides information as to the type of variable.

```
PS C:\Users>
PS C:\Users> $i = 0
PS C:\Users> $i.GetType()
IsPublic IsSerial Name                                     BaseType
----- ----- ----
True     True    Int32                                  System.ValueType

PS C:\Users> $r = 1.10
PS C:\Users> $r.GetType()
IsPublic IsSerial Name                                     BaseType
----- ----- ----
True     True    Double                                 System.ValueType

PS C:\Users> $s = 'The quick brown fox'
PS C:\Users> $s.GetType()
IsPublic IsSerial Name                                     BaseType
----- ----- ----
True     True    String                                System.Object

PS C:\Users>
```

In the above illustration, `$i` is a 32 bit integer. `$r` is a double precision floating point number (usually used when very high precision is required). `$s` is text string.

More information on variables, types, and methods is provided in later sections.

Piping & Redirection

Sunday, January 13, 2013
12:13 PM

Input and output redirection have been available in operating systems since the inception of

Unix. Typically commands in the Unix or Linux shell accept input from `stdin` (standard input) which is the console (keyboard). Input redirection allows the input to come from a text file instead of the console (keyboard). Similarly the default output destination for command in a *nix shell is `stdout` (standard output)*

which is also the console (monitor). Output redirection diverts output from stdout to a text file.

The Windows cmd shell adopted the very basic redirection functionality from Unix. The example below demonstrates output redirection. The output redirection operator > redirects output destined for the console to the designated file, out.txt in the example below.

```
C:\Users>
C:\Users> echo This is displayed on the console
This is displayed on the console
C:\Users> echo this is written to the file > out.txt
C:\Users> type out.txt
this is written to the file
```

The next example illustrates input redirection using the find command. The find operator searches for a text string within a series of text string. In the example below, find is looking for the string "red" in the lines entered from the console. Once the matching substring is found, find displays the line on the console. In the illustration, the lines underlined in red are the output displayed by find.

```
C:\Users>
C:\Users> find "red"
The quick green fox
the quick blue fox
the red fox
the red fox
the purple fox
the slow fox
the very quick red fox
the very quick red fox
^Z
```

Creating a text file called in.txt using the same lines of text, we can use the input redirection operator < to redirect the input from the console to in.txt.

```
C:\Users>
C:\Users> type in.txt
The quick green fox
the quick blue fox
the red fox
the purple fox
the slow fox
the very quick red fox

C:\Users> find "red" < in.txt
the red fox
the very quick red fox
```

Piping is the natural progression from redirection. Piping allows output of a command to *piped*, provided as input, to another command. The piping operator is the vertical bar |. In the illustration below, using the legacy cmd shell, the host 8.8.8.8 is pinged one time; the output of the ping command is displayed in the console window.

```
C:\Users>
C:\Users>ping 8.8.8.8 -n 1

Pinging 8.8.8.8 with 32 bytes of data:
Reply from 8.8.8.8: bytes=32 time=25ms TTL=48

Ping statistics for 8.8.8.8:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 25ms, Maximum = 25ms, Average = 25ms
```

The image below demonstrates the use of piping in the cmd shell. The output of ping is piped into find. The find command displays the matching line(s) of text that contain the string "TTL".

```
C:\Users>
C:\Users>ping 8.8.8.8 -n 1 | find "TTL"
Reply from 8.8.8.8: bytes=32 time=16ms TTL=48
```

PowerShell provides a more robust piping capability. Below in the PowerShell console, we pipe the output of ping into the cmdlet select-string to select the one

line that contains the text string "TTL". This not only demonstrates piping but illustrates the use of legacy commands with cmdlets.

```
PS C:\Users>
PS C:\Users> ping 8.8.8.8 -n 1 | select-string 'TTL'
Reply from 8.8.8.8: bytes=32 time=16ms TTL=48
```

Because piping is much more powerful in the PowerShell than legacy shells, input redirection is not available. PowerShell provides only output direction. However PowerShell cmdlets like get-content and out-host along with piping render both input and output redirection unnecessary.

In the illustration below, the get-content cmdlet reads the contents of in.txt into memory. The contents are piped line by line to the where-object cmdlet. This cmdlet tests the current object (\$_) in the pipeline, which is just the current line of text, to determine whether the line contains the text string "red". The wildcards (<https://technet.microsoft.com/en-us/library/bb490639.aspx>) * preceding the match pattern match any number of characters.

```
PS C:\Users>
PS C:\Users> get-content in.txt
The quick green fox
the quick blue fox
the red fox
the purple fox
the slow fox
the very quick red fox
PS C:\Users>
PS C:\Users> get-content in.txt | where-object { $_ -like '*red*' }
the red fox
the very quick red fox
PS C:\Users>
```

Variables may also be used with the legacy DOS commands and with piping. In the next example, an IP address is assigned to a variable \$IP. The variable is then used in the ping command. In the example below \$ip holds a single datum which is the string '8.8.8.8'. It's important to understand that ping doesn't understand variables. PowerShell takes the value contained in \$ip and *passes* it to ping.

```
PS C:\Users>
PS C:\Users> $ip = '8.8.8.8'
PS C:\Users> ping $ip

Pinging 8.8.8.8 with 32 bytes of data:
Reply from 8.8.8.8: bytes=32 time=69ms TTL=44
Reply from 8.8.8.8: bytes=32 time=72ms TTL=44
Reply from 8.8.8.8: bytes=32 time=67ms TTL=44
Reply from 8.8.8.8: bytes=32 time=85ms TTL=44

Ping statistics for 8.8.8.8:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 67ms, Maximum = 85ms, Average = 73ms
PS C:\Users>
```

What happens multiple addresses assigned to the variable \$ip? Since ping is expecting only one address, an error is thrown "Bad parameter..". The reason for this error is becomes obvious by replacing the variable in the command with the value contained in \$ip. Doing this results in the command "ping 8.8.8.8, 8.8.4.4, 207.75.134.1" which is clearly invalid because ping is only expecting one value to be passed.

In this case, \$ip is a type of variable called an *array* (also referred to as a collection). Notice the display when \$ip is displayed on the console, each element in the array id displayed on separate lines.

```
PS C:\Users>
PS C:\Users> $ip = '8.8.8.8','8.8.4.4','207.75.134.1'
PS C:\Users>
PS C:\Users> ping $ip
Bad parameter 8.8.4.4.
PS C:\Users>
PS C:\Users> $ip
8.8.8.8
8.8.4.4
207.75.134.1
PS C:\Users>
```

To ping all addresses contained in \$ip we have to create a pipeline. The temptation is to create the one shown below, which gives some unintended results. This doesn't work because of the way PowerShell handles objects coming down the pipeline.

```

PS C:\Users>
PS C:\Users> $ip = '8.8.8.8','8.8.4.4','207.75.134.1'
PS C:\Users>
PS C:\Users> $ip | ping $_

Usage: ping [-t] [-a] [-n count] [-l size] [-f] [-i TTL] [-v TOS]
           [-r count] [-s count] [[-j host-list] | [-k host-list]]
           [-w timeout] [-R] [-S srcaddr] [-4] [-6] target_name

Options:
  -t          Ping the specified host until stopped.
              To see statistics and continue - type Control-Break;
              To stop - type Control-C.
  -a          Resolve addresses to hostnames

```

The above pipeline may be corrected using the *foreach-object* cmdlet. This cmdlet allows PowerShell to individually process each object moving through the pipeline. By using foreach-object, ping is presented with one IP address instead of an array of IP addresses.

```

PS C:\Users>
PS C:\Users> $ip = '8.8.8.8','8.8.4.4','207.75.134.1'
PS C:\Users>
PS C:\Users> $ip | foreach-object { ping $_ }

Pinging 8.8.8.8 with 32 bytes of data:
Reply from 8.8.8.8: bytes=32 time=60ms TTL=44

Ping statistics for 8.8.8.8:
  Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
  Approximate round trip times in milli-seconds:
    Minimum = 60ms, Maximum = 60ms, Average = 60ms

Pinging 8.8.4.4 with 32 bytes of data:
Reply from 8.8.4.4: bytes=32 time=60ms TTL=44

Ping statistics for 8.8.4.4:
  Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
  Approximate round trip times in milli-seconds:
    Minimum = 60ms, Maximum = 60ms, Average = 60ms

Pinging 207.75.134.1 with 32 bytes of data:
Reply from 207.75.134.1: bytes=32 time=54ms TTL=237
Reply from 207.75.134.1: bytes=32 time=54ms TTL=237

```

For the sake of reducing verbosity of the output, we add the -n switch to cause ping to ping only once and pipe the output of ping into select-string to select only the lines that contain the substring 'TTL'. Note, we may also use where-object as in the above to accomplish the same outcome.

```
PS C:\Users>
PS C:\Users> $ip = '8.8.8.8','8.8.4.4','207.75.134.1'
PS C:\Users>
PS C:\Users> $ip | foreach-object { ping $_ -n 1 } | select-string 'TTL'
Reply from 8.8.8.8: bytes=32 time=60ms TTL=44
Reply from 8.8.4.4: bytes=32 time=60ms TTL=44
Reply from 207.75.134.1: bytes=32 time=54ms TTL=237
```

To complete this example, we replace the legacy ping command with a cmdlet test-connection.

```
PS C:\Users>
PS C:\Users> $ip = '8.8.8.8','8.8.4.4','207.75.134.1'
PS C:\Users>
PS C:\Users> $ip | foreach-object { test-connection $_ -count 1 }

Source      Destination      IPV4Address      IPV6Address
----      -----      -----
EL-CID      8.8.8.8      8.8.8.8
EL-CID      8.8.4.4      8.8.4.4
EL-CID      207.75.134.1      207.75.134.1      207.75.134.1

PS C:\Users>
```

Objects

Sunday, January 13, 2013
12:08 PM

Historically, Unix/Linux scripting languages manipulated text output from commands. While these are powerful scripting environments, the designers of PowerShell understood the shortcomings of this model.

PowerShell takes a completely different approach. All PowerShell cmdlets emit *objects*. An object is a datum that has:

- A certain *state* that is described by **properties**
- Behaviors that are exposed by **methods**

Objects emitted by cmdlets may be manipulated using the properties of the object. The object state may be altered by using the methods.

Let's look at an problem of extracting the TTL for a ping request. Below is a ping the cmd shell. The problem is to extract the value 58 from the output.

```
C:\Users>
C:\Users>
C:\Users>ping 8.8.8.8 -n 1

Pinging 8.8.8.8 with 32 bytes of data:
Reply from 8.8.8.8: bytes=32 time=56ms TTL=58

Ping statistics for 8.8.8.8:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 56ms, Maximum = 56ms, Average = 56ms
```

The bash script below is one solution to the problem. to determine the ttl of the host 8.8.8.8 would appear similar to that below:

```
ping -c1 8.8.8.8 | grep ttl | awk -F '=' '{ print $3}' | awk -F '' '{print $1}'
```

```
mgalea@el-cid:~$ ping -c1 8.8.8.8 | grep ttl | awk -F '=' '{ print $3}' | awk -F '' '{print $1}'
58
mgalea@el-cid:~$
```

For the uninitiated the above code is very intimidating and not readily understood. Solving this problem in PowerShell requires using Test-Connection cmdlet which outputs an ping status object as shown below. Only the default properties are display on the console. The ping status object has many more

properties including the ResponseTimetoLive which is the TTL set by the destination host.

```
PS C:\Users>
PS C:\Users> test-connection 8.8.8.8 -count 1
Source        Destination      IPV4Address      IPV6Address
----        -----      -----
EL-CID        8.8.8.8        8.8.8.8
By
--32
PS C:\Users>
```

The solution to the problem is shown below. The ping status object is stored in the variable \$result. The write-host cmdlet is then used to display the value of the ResponseTimeToLive property of the object contained in \$result.

```
$result = Test-Connection 8.8.8.8 -count 1
Write-Host $result.ResponseTimeToLive
```

```
PS C:\Users>
PS C:\Users> $result = Test-Connection 8.8.8.8 -count 1
PS C:\Users> Write-Host $result.ResponseTimeToLive
58
PS C:\Users>
```

The above script could be enhanced further to provide a more informative display.

```
$ip = "8.8.8.8"
$result = Test-Connection $ip -count 1
Write-Host "The TTL returned by $ip is
 $($result.ResponseTimeToLive)"
```

```
PS C:\Users>
PS C:\Users> $ip = "8.8.8.8"
PS C:\Users> $result = Test-Connection $ip -count 1
PS C:\Users> Write-Host "The TTL returned by $ip is $($result.ResponseTimeToLive)"
The TTL returned by 8.8.8.8 is 58
PS C:\Users>
```

We could expand the above example to ping multiple hosts and show different properties of the ping status object by using the format-table cmdlet.

```

PS C:\Users>
PS C:\Users> $ip = '8.8.8.8','8.8.4.4','207.75.134.1'
PS C:\Users>
PS C:\Users> $ip | foreach-object { test-connection $_ -count 1 } | format-table address,
address      replysize responsetimetolive
-----
8.8.8.8          32            54
8.8.4.4          32            56
207.75.134.1     32           242

PS C:\Users>

```

In the above examples, the ping status object is derived from a pre-defined template called a *class*. The class identifies the properties and the methods of the object. Every object is *instantiated*, i.e., created, from its class. Somewhere in the Windows Operating System either in the .Net framework, WMI, or other locations are the classes that describe the objects used by PowerShell.

Class Example:

To understand the notion of a class, let's make a real life analogy of classes and objects using a dog. A dog has certain attributes (properties) such as breed, color, weight, etc. A dog also has certain behaviors: barks, runs, etc. In the example below we create a dog class using C# with the some properties and methods common to a dog. We use the Add-Type cmdlet to add the "aDog" class to the PowerShell script.

```

<# We use C# code to create a class called aDog. The Add-Type cmdlet
takes the C# code defined in the here string (the text delimited by @@
and '@'), compiles it and creates the class #>

Add-Type @@
public class aDog
{
    public string color;
    public string breed;

    public string bark(int ntimes) {
        string woof = "";
        for (int i = 1; i <= ntimes; i++)
        {

```



```

PS C:\Users>
PS C:\Users> <# We use C# code to create a class called aDog. The Add-Type cmdlet takes
string (the text delimited by '@' and '@'), compiles it and creates the class #>
PS C:\Users>
PS C:\Users> Add-Type @'
>> public class aDog
>> {
>>     public string color;
>>     public string breed;
>>
>>     public string bark(int ntimes) {
>>         string woof = "";
>>         for (int i = 1; i <= ntimes; i++)
>>         {
>>             woof = woof + "Woof!  ";
>>         }
>>         return woof;
>>     }
>>
>> }
>> @@
PS C:\Users> $lab = New-Object aDog
# and store it in the variable $lab
PS C:\Users> $lab.breed = 'Labrador'
PS C:\Users> $lab.color = 'Chocolate'
PS C:\Users> $speak = new-object -com SAPI.SPvoice
PS C:\Users> $speak.speak( $lab.bark(4) )
1
PS C:\Users> $lab
color      breed
-----      -----
Chocolate  Labrador

PS C:\Users>

```

Determining the properties and methods of an object:

The question at this point is how does one know the properties and methods of the objects. The get-member cmdlet exposes the properties and methods of an object. The object in question may be *piped* into get-member as shown below. We will use get-member extensively to discover properties and methods of the objects returned by cmdlets.

```
$result = Test-Connection 8.8.8.8 -count 1
$result | get-member
```

```

PS C:\>
PS C:\> $result = Test-Connection 8.8.8.8 -count 1
PS C:\> $result | get-member

TypeName: System.Management.ManagementObject#root\cimv2\win32_PingStatus

Name          MemberType      Definition
----          -----
PSCo...       AliasProperty PSComputerName = __SERVER
Address       Property        string Address {get;set;}
BufferSize    Property        uint32 BufferSize {get;set;}
NoFragmentation Property      bool NoFragmentation {get;set;}
PrimaryAddressResolutionStatus Property uint32 PrimaryAddressResolutionStatus {get;set;}
ProtocolAddress   Property      string ProtocolAddress {get;set;}
ProtocolAddressResolved Property string ProtocolAddressResolved {get;set;}
RecordRoute    Property      uint32 RecordRoute {get;set;}
ReplyInconsistency Property  bool ReplyInconsistency {get;set;}
ReplySize     Property      uint32 ReplySize {get;set;}
ResolveAddressNames Property  bool ResolveAddressNames {get;set;}
ResponseTime   Property      uint32 ResponseTime {get;set;}
ResponseTimeToLive Property  uint32 ResponseTimeToLive {get;set;}
RouteRecord    Property      string[] RouteRecord {get;set;}
RouteRecordResolved Property string[] RouteRecordResolved {get;set;}
SourceRoute    Property      string SourceRoute {get;set;}
SourceRouteType Property     uint32 SourceRouteType {get;set;}
StatusCode     Property      uint32 StatusCode {get;set;}
Timeout       Property      uint32 Timeout {get;set;}
TimeStampRecord Property string[] TimeStampRecord {get;set;}
TimeStampRecordAddress Property string[] TimeStampRecordAddress {get;set;}
TimeStampRecordAddressResolved Property string[] TimeStampRecordAddressResolved {get;set;}
TimestampRoute Property     uint32 TimestampRoute {get;set;}
TimeToLive     Property      uint32 TimeToLive {get;set;}
TypeofService  Property     uint32 TypeofService {get;set;}
__CLASS        Property      string __CLASS {get;set;}
__DERIVATION   Property      string[] __DERIVATION {get;set;}
__DYNASTY     Property      string __DYNASTY {get;set;}
__GENUS        Property     int __GENUS {get;set;}
__NAMESPACE   Property      string __NAMESPACE {get;set;}
__PATH         Property      string __PATH {get;set;}
__PROPERTY_COUNT Property     int __PROPERTY_COUNT {get;set;}
__RELPATH      Property      string __RELPATH {get;set;}
__SERVER       Property      string __SERVER {get;set;}
__SUPERCLASS   Property      string __SUPERCLASS {get;set;}
ConvertFromDateTime ScriptMethod System.Object ConvertFromDateTime()
Convert.ToDateTime ScriptMethod System.Object Convert.ToDateTime()
IPV4Address    ScriptProperty System.Object IPV4Address {get=$iphost}
IPV6Address    ScriptProperty System.Object IPV6Address {get=$iphost}

```

We inspect get-member, in the case the alias gm, to inspect the properties and methods of the \$slab variable and the \$speak variable.

```

PS C:\Users>
PS C:\Users> $lab | gm

    TypeName: aDog

Name      MemberType Definition
----      -----
bark     Method     string bark(int ntimes)
Equals   Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType   Method     type GetType()
ToString  Method     string ToString()
breed    Property   string breed {get;set;}
color    Property   string color {get;set;}

PS C:\Users> $speak | gm

    TypeName: System.__ComObject#{269316d8-57bd-11d2-9eee-00c04f797396}

Name      MemberType Definition
----      -----
DisplayUI Method     void DisplayUI (int, string, string, Variant)
GetAudioOutputs Method     ISpeechObjectTokens GetAudioOutputs (string)
GetVoices   Method     ISpeechObjectTokens GetVoices (string, string)
IsUISupported Method     bool IsUISupported (string, Variant)
Pause      Method     void Pause ()
Resume    Method     void Resume ()
Skip       Method     int Skip (string, int)
Speak      Method     int Speak (string, SpeechVoiceSpeakFlags)
SpeakCompleteEvent Method     int SpeakCompleteEvent ()
SpeakStream Method     int SpeakStream (ISpeechBaseStream, SpeechVoice)
WaitUntilDone Method     bool WaitUntilDone (int)
AlertBoundary Property   SpeechVoiceEvents AlertBoundary () {get}
AllowAudioOutputFormatChangesOnNextSet Property   bool AllowAudioOutputFormatChangesOnNextSet ()
AudioOutput  Property   ISpeechObjectToken AudioOutput () {get}
AudioOutputStream Property   ISpeechBaseStream AudioOutputStream () {get}
EventInterests Property   SpeechVoiceEvents EventInterests () {get}
Priority    Property   SpeechVoicePriority Priority () {get} {set}
Rate        Property   int Rate () {get} {set}
Status      Property   ISpeechVoiceStatus Status () {get}
SynchronousSpeakTimeout Property   int SynchronousSpeakTimeout () {get} {set}
Voice       Property   ISpeechObjectToken Voice () {get} {set}
Volume     Property   int Volume () {get} {set}

PS C:\Users>

```

File System Objects:

Let's take a look at some practical uses of objects in PowerShell using file system objects, i.e., files and directories. We are all familiar with the properties of a file:

name	Identifies the file
size	File size in bytes
date modified	The date-time the file was last modified
creation time	The date-time the file was created
	<i>Many more</i>

Some of the actions we can take on a file are shown below:

copy	Copy the file or directory to some other location
move	Move the file or directory to some other location
delete	Delete the file or directory
	<i>Many more</i>

The traditional ways of copy and deleting a file using the cmd shell is shown below.

```
C:\Users>
C:\Users> copy a.txt b.txt
      1 file(s) copied.

C:\Users>
C:\Users> del b.txt

C:\Users>
```

PowerShell also includes cmdlets that perform these actions in the traditional manner.

```
PS C:\users>
PS C:\users> copy-item a.txt .\b.txt
PS C:\users>
PS C:\users> remove-item .\b.txt
PS C:\users>
```

Unique to PowerShell are the ways for executing the above actions in a non-traditional way.

In the illustration below the variable \$f holds the *FileInfo* object that references the file a.txt.

```
$f = get-childitem a.txt # get the fileInfo object that references  
# a.txt  
$f.copyto('.\b.txt')      # copy a.txt as b.txt to the same directory  
$f.delete()              # delete a.txt
```

```
PS C:\users>  
PS C:\users> $f = get-childitem a.txt  
PS C:\users> $f.copyto('.\b.txt')  


| Mode  | LastWriteTime    | Length | Name  |
|-------|------------------|--------|-------|
| -a--- | 5/8/2015 2:49 PM | 12     | b.txt |

  
PS C:\users> $f.delete()  
PS C:\users>
```

One final note on cmdlets copy-item, delete-item, and many other cmdlets that operate of files and directories. In the cmd shell, copying or deleting a directory requires different command than those used for files. Not so for PowerShell, the copy-item and delete-item operate on both files and directories and, as we will see later, this cmdlets work with many different types of targets including file system objects, registry items, and many others.

Creating a text file using .Net objects

While we generally will not be using .NET classes directly, the following script illustrates the ease of using .NET classes. The script below uses the *FileInfo* .NET class to create a text file called z.txt and write three lines to it.

```
$f = New-Object System.IO.FileInfo # create a file info object  
z.txt  
$sw = $f.createText()           # create the text file z.txt in  
# the current directory then create  
# a streamwriter associated with  
# the file
```

```
$sw.WriteLine('Line 1')          # Use WriteLine method of the
                                # streamwriter object to write the
                                # first line the file
$sw.WriteLine('Line 2')          # Write the second line
$sw.WriteLine('Line 3')          # Write the third line
$sw.Close()                     # Close the file
```

The illustration below shows the results of running this script in the ISE.

The screenshot shows the PowerShell ISE interface. The top window is titled "psex1.ps1" and contains the PowerShell script. The bottom window is a terminal window showing the execution of the script and its output.

```
psex1.ps1 x
1 $f = New-Object System.IO.FileInfo "c:\users\z.txt" # create a file
2 $sw = $f.CreateText()      # create the text file z.txt in the current directory
3 $sw.WriteLine('Line 1')    # Write the text into the file
4 $sw.WriteLine('Line 2')    # Write a second line to textdir
5
6 $sw.WriteLine('Line 3')    # Write the third line
7 $sw.Close()               # Close the file
8
```

```
PS C:\Users>
PS C:\Users> C:\Users\psex1.ps1

PS C:\Users> get-content .\z.txt
Line 1
Line 2
Line 3

PS C:\Users> |
```

Scripts

Saturday, December 23, 2017
11:15 AM

Scripts are text files with a .ps1 extension that contain cmdlets, pipeline, and PowerShell statements. Scripts may be created using Notepad, another text editor, or the Integrated Script Environment (ISE). The ISE may be the most convenient since it always saves the script file with a .ps1 extension.

The script below pings a series of hosts using the test-connection cmdlet and shows the destination IP address and the status. A status code of 0 indicates a successful ping.

```
write-host
$ip = '8.8.8.8', '8.8.4.4', '207.75.134.1'
write-host "Pinging hosts $ip"
$ip | foreach-object { test-connection $_ -count 1}
    | format-table ProtocolAddress, StatusCode
```

```
PS>
PS>
PS> get-content .\pinger.ps1
#clear-host
write-host
$ip = '8.8.8.8', '8.8.4.4', '207.75.134.1'
write-host "Pinging hosts $ip"
$ip | foreach-object { test-connection $_ -count 1} | format-table ProtocolAddress, StatusCode
PS>
PS> .\pinger.ps1
Pinging hosts 8.8.8.8 8.8.4.4 207.75.134.1
ProtocolAddress StatusCode
----- -----
8.8.8.8          0
8.8.4.4          0
207.75.134.1    0
```

Execution Policy

Wednesday, July 31, 2013
2:30 PM

get-help about_execution_policies

Historically, i.e., prior to 2006, Windows scripting was done using VBScript or Jscript and the Windows Scripting Host (WSH). The environment was notoriously insecure. This resulted in many exploits written in these languages.

Understanding the weakness of previous scripting environments, PowerShell was developed with security in mind. Providing total security is not feasible since it requires not executing scripts, PowerShell provides a layered script execution policy that the administrator may implement. The execution policy ranges from allowing unrestricted execution of any PowerShell script to requiring that all scripts be digitally signed:

Restricted - This is the default policy. No scripts can be run. PowerShell may be used only in interactive mode.

AllSigned - Only scripts signed by a trusted publisher can be run.

RemoteSigned - Downloaded scripts must be signed by a trusted publisher before they can be run.

Unrestricted - No restrictions; all Windows PowerShell scripts can be run

The default policy "Restricted" means that the PowerShell console may be executed and cmdlets and pipelines entered interactively. But attempt to run a script results in an error. The execution policy must be change from the default as shown below. For home use, the safe policy is "RemotedSigned".

The execution policy may be set by executing the cmdlet `set-executionpolicy` which must be executed in a console running under administrative privileges. In the examples below note the window title.

```
powershell - Shortcut
PS C:\Users> Set-ExecutionPolicy RemoteSigned
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the
execution policy might expose you to the security risks described in the
about_Execution_Policies help topic at http://go.microsoft.com/fwlink/?LinkID=135170. Do you
want to change the execution policy?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): y
Set-ExecutionPolicy : Access to the registry key 'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerSh
(LocalMachine) scope, start Windows PowerShell with the "Run as administrator" option. To chang
At Line:1 char:1
+ Set-ExecutionPolicy RemoteSigned
+ ~~~~~
+ CategoryInfo          : PermissionDenied: () [Set-ExecutionPolicy], UnauthorizedAccessEx
+ FullyQualifiedErrorId : System.UnauthorizedAccessException,Microsoft.PowerShell.Commands.
PS C:\Users>
```

```
Administrator: powershell - Shortcut
PS C:\users> Set-ExecutionPolicy RemoteSigned
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the
execution policy might expose you to the security risks described in the
about_Execution_Policies help topic at http://go.microsoft.com/fwlink/?LinkID=135170. Do you
want to change the execution policy?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): y
PS C:\users> -
```

Another feature implemented to avoid executing a malicious script in the current directory, PowerShell requires that script names be prefixed by .\ to indicate the current directory

The rationale for this requirement is based upon how Windows searches for executables (.exe, .bat, .vbs, etc.) and how this was exploited by hackers. When a command name, i.e., notepad.exe, ipconfig.exe, etc., is entered on the command

line, Windows first searches in the current directory for the executable matching the command name. If the executable is not found, Windows then uses the Path environment variable and searches the directories listed in this variable for the executable.

In the past a common hacker technique was to cause the naïve user to download a malicious executable under an innocuous name, e.g., notepad.exe, to the user's current directory. The naïve user would attempt to execute Notepad by entering notepad at the command line. Given the Windows search behavior described above, the malicious executable would execute.

```
PS C:\users>
PS C:\users> hello.ps1
hello.ps1 : The term 'hello.ps1' is not recognized as the name of a cmdlet, function, script
path is correct and try again.
At line:1 char:1
+ hello.ps1
+
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (hello.ps1:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

Suggestion [3,General]: The command hello.ps1 was not found, but does exist in the current lo
ommend, instead type ".\hello.ps1". See "get-help about_Command_Precedence" for more details.
PS C:\users>
```

By enforcing to use of ".\" syntax for executing a script in the current directory, PowerShell is forcing the user to make a conscious declaration that they want to execute the script in the current directory.

```
PS C:\users>
PS C:\users> .\hello.ps1
```

```
Hello world
```

```
PS C:\users>
```

Media

Saturday, August 8, 2015
8:01 AM

Chapter 1 - Customizing the Shell

<https://www.youtube.com/watch?v=6CRTahGYnws&list=PL6D474E721138865A>

PowerShell Console - Properties

<https://www.youtube.com/watch?v=EJQjPUVrxzM>

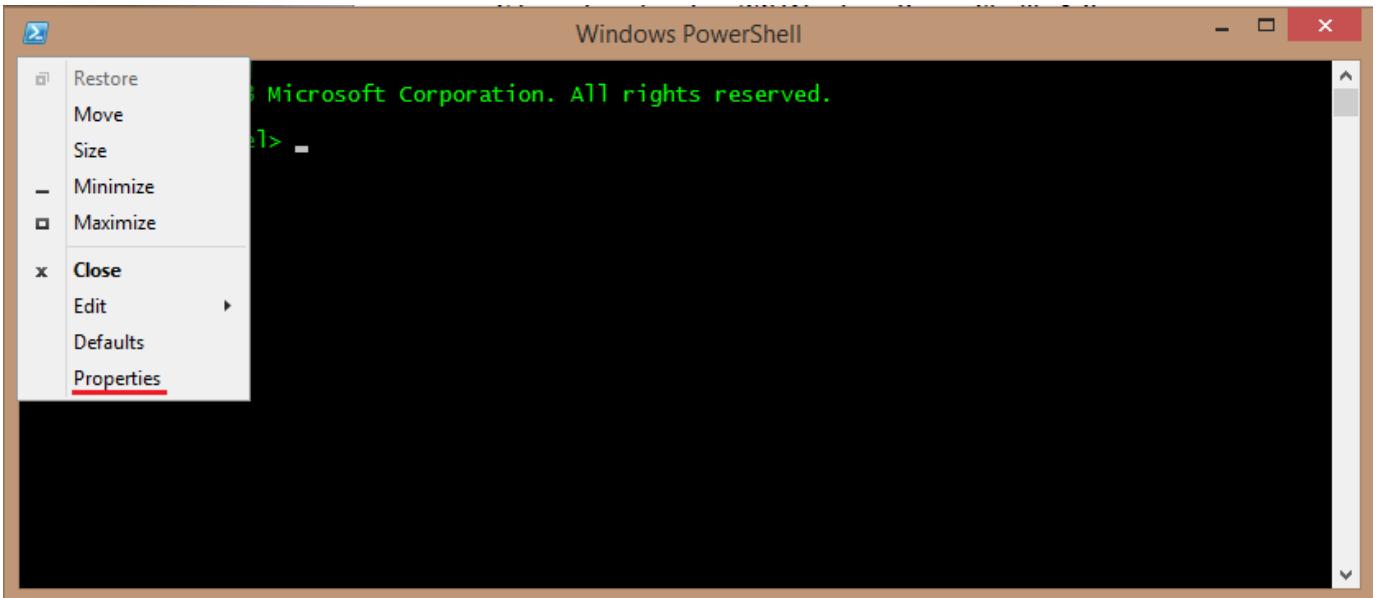
Console Configuration

Sunday, January 13, 2013
12:11 PM

Customizing the Console Window

In the past for most administrators, working in the cmd window was common but not necessarily frequent. Because PowerShell provides robust administrative features and has become the administrative platform for Windows, administrators will find themselves working in the PowerShell console much more frequently. So knowing how to customize the console window for ease of use is important.

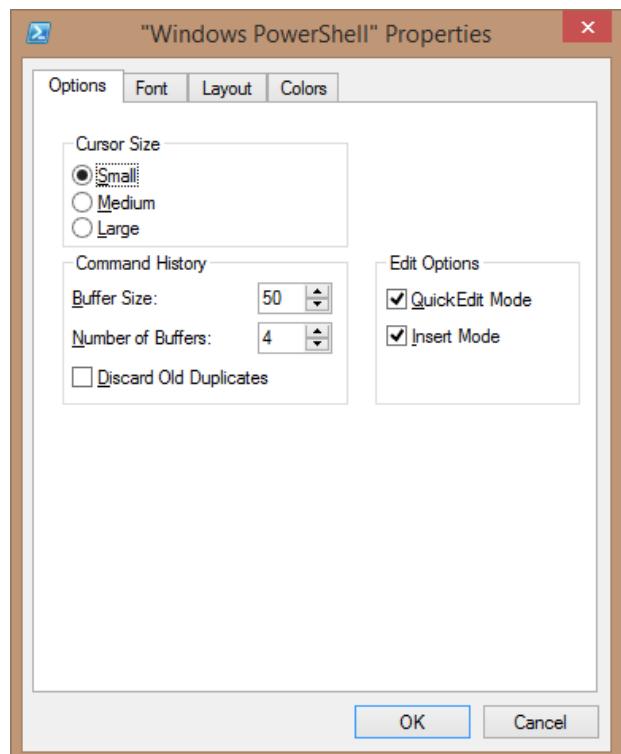
Customizing the console is no different than customizing the cmd window, click on the PowerShell icon in the top left corner of the window to expose a shortcut menu. Select properties in the menu.



Modifying the properties of the console window is generally a good idea since the default settings are usually inadequate. To inspect and modify the properties of the console window, click on the PowerShell icon in the top left corner of the window to open a shortcut menu. Select Properties from the shortcut menu.

The first property to modify the "Quick Edit Mode". This property should be set to allow left-dragging over text in the console to select it. If this property is not set, selecting text requires opening the shortcut menu for the console as described above, select Mark, then left-drag over the desired text to select it.

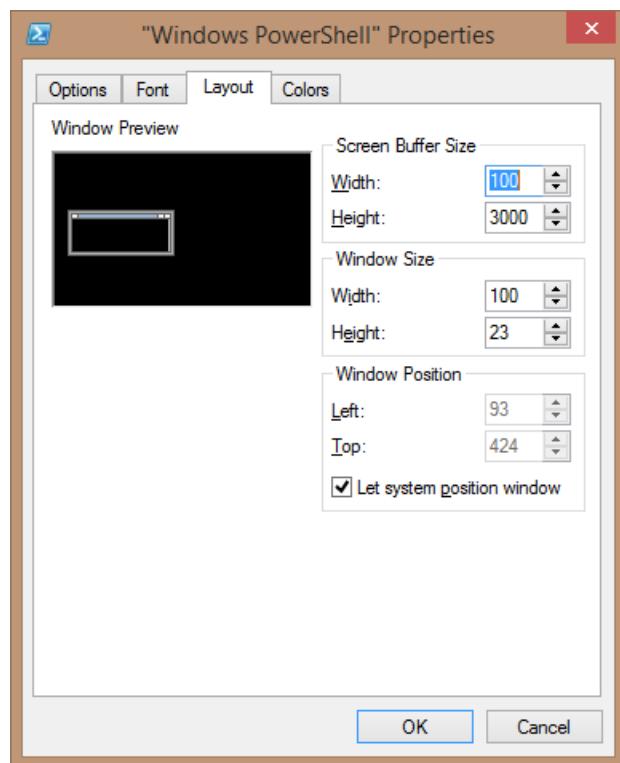
To copy the text to the clipboard for subsequent paste in another app, requires pressing the enter key with the desired text selected.



Altering the Screen Buffer Size property is also a good idea. In the image on the right, the width and height are changed to 100 characters wide and 3000 lines of history displayed respectively. These settings allow scrolling back the window to view previously entered cmdlets and output.

Setting the font and font sizes in the Fonts tab are a matter of personal preference.

The Colors tab allows customization of the window background color, font color, and other colors. These, like fonts, is a matter of personal preference.



Customizing the PowerShell Console

Many of the window properties described above may be altered dynamically by using PowerShell features.

The Get-Host cmdlet displays the console properties.

```
PS C:\Users>
PS C:\Users> get-host

Name          : ConsoleHost
Version       : 4.0
InstanceId    : 9e6c2962-903b-4781-8e3e-c2e18a127942
UI            : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture : en-US
CurrentUICulture : en-US
PrivateData   : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
IsRunspacePushed : False
Runspace      : System.Management.Automation.Runspaces.LocalRunspace
```

These properties are found in the PowerShell automatic variable \$host.

```

PS C:\Users>
PS C:\Users> $host

Name          : ConsoleHost
Version       : 4.0
InstanceId    : 9e6c2962-903b-4781-8e3e-c2e18a127942
UI           : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture : en-US
CurrentUICulture : en-US
PrivateData   : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
IsRunspacePushed : False
Runspace      : System.Management.Automation.Runspaces.LocalRunspace

```

Since all variables in PowerShell are objects we may reveal the properties and methods of the \$host object using get-member. The UI property allows us to customize the *user interface* dynamically. Look at the definition UI we see that it is not just a simple property but an object.

```

PS C:\Users>
PS C:\Users> $host | get-member

TypeName: System.Management.Automation.Internal.Host.InternalHost

Name          MemberType Definition
----          ----
EnterNestedPrompt Method void EnterNestedPrompt()
Equals        Method bool Equals(System.Object obj)
ExitNestedPrompt Method void ExitNestedPrompt()
GetHashCode   Method int GetHashCode()
GetType       Method type GetType()
NotifyBeginApplication Method void NotifyBeginApplication()
NotifyEndApplication Method void NotifyEndApplication()
PopRunspace    Method void PopRunspace(), void IHostSupportsInteractiveSession.PopRunspace()
PushRunspace   Method void PushRunspace(runspace runspace), void IHostSupportsInteractiveSession.PushRunspace()
SetShouldExit  Method void SetShouldExit(int exitCode)
ToString      Method string ToString()
CurrentCulture Property cultureinfo CurrentCulture {get;}
CurrentUICulture Property cultureinfo CurrentUICulture {get;}
InstanceId     Property guid InstanceId {get;}
IsRunspacePushed Property bool IsRunspacePushed {get;}
Name          Property string Name {get;}
PrivateData   Property psobject PrivateData {get;}
Runspace      Property runspace Runspace {get;}
UI           Property System.Management.Automation.Host.PSHostUserInterface UI {get;}
Version       Property version Version {get;}

```

We reveal the components of the \$host.UI property using get-member. The only property of the UI object is RawUI which is just another object.

TypeName: System.Management.Automation.Internal.Host.InternalHostUserInterface		
Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
Prompt	Method	System.Collections.Generic.Dictionary[string,psobj]
PromptForChoice	Method	int PromptForChoice(string caption, string message)
PromptForCredential	Method	pscredential PromptForCredential(string caption, s
ReadLine	Method	string ReadLine()
ReadLineAsSecureString	Method	securestring ReadLineAsSecureString()
Tostring	Method	string ToString()
Write	Method	void Write(string value), void Write(System.Consol
WriteDebugLine	Method	void WriteDebugLine(string message)
WriteErrorLine	Method	void WriteErrorLine(string value)
WriteLine	Method	void WriteLine(), void WriteLine(string value), vo
WriteProgress	Method	void WriteProgress(long sourceId, System.Manageme
WriteVerboseLine	Method	void WriteVerboseLine(string message)
WriteWarningLine	Method	void WriteWarningLine(string message)
RawUI	Property	System.Management.Automation.Host.PSHostRawUserInt

The RawUI property of the UI (user interface) property allows us to view and configure the appearance of the console.

```
PS C:\Users>
PS C:\Users> $host.ui.rawui

ForegroundColor      : Green
BackgroundColor       : Black
CursorPosition       : 0,2
WindowPosition       : 0,0
CursorSize           : 25
BufferSize           : 120,3000
WindowSize           : 120,50
MaxWindowSize        : 120,81
MaxPhysicalWindowSize: 213,81
KeyAvailable         : False
WindowTitle          : Windows PowerShell
```

The UI console configuration may be changed by altering the properties of \$host.ui.rawui object. Changes made in the manner below only persist for the current session. A permanent custom console configuration may be accomplished by setting \$host.ui.rawui properties in a profile script (more in a later section).

The background and foreground color are set using the commands below. The clear-host cmdlet clears the console window so that that change is consistent in the window.

```
$host.ui.rawui.BackgroundColor = 'darkgray'  
$host.ui.rawui.ForegroundColor = 'cyan'  
$host.ui.rawui.WindowTitle = 'My PowerShell Console'  
clear-host
```

```
PS C:\Users>  
PS C:\Users> $host.ui.rawui.BackgroundColor = 'darkgray'  
PS C:\Users> $host.ui.rawui.ForegroundColor = 'cyan'  
PS C:\Users> $host.ui.rawui.WindowTitle = 'My PowerShell Console'  
PS C:\Users> clear-host
```

In the above illustration, the clear-host cmdlet was not execute to demonstrate the need for the clear-host cmdlet.

Changing the multi-valued properties such as BufferSize, WindowSize, etc. is not a simple matter of assigning new values because these properties are not string types. The type of each of these properties may be determined by piping the property into get-member.

```
PS C:\Users>  
PS C:\Users> $host.UI.RawUI.WindowSize | gm  
  
 TypeName: System.Management.Automation.Host.Size  
  
Name      MemberType  Definition  
----      ----  
Equals    Method     bool Equals(System.Object obj)  
GetHashCode Method     int GetHashCode()  
GetType   Method     type GetType()  
ToString  Method     string ToString()  
Height    Property   int Height {get;set;}  
Width    Property   int Width {get;set;}  
  
PS C:\Users> $host.UI.RawUI.WindowSize  
Width  
----  
120  
Height  
----  
50
```

Knowing the type, the property may be modified by creating an object of that type and assigning the appropriate values at creation time. The code below demonstrates the steps necessary to change the window size dynamically.

```
$uiproperty = $host.ui.rawui.WindowSize  
$uiproperty.Height = 30  
$uiproperty.Width = 110  
$host.ui.rawui.WindowSize = $uiproperty
```

After executing the above, the window size has been changed as shown below.

```
PS C:\Users>  
PS C:\Users> $uiproperty = $host.ui.rawui.WindowSize  
PS C:\Users> $uiproperty.Height = 30  
PS C:\Users> $uiproperty.Width = 110  
PS C:\Users> $host.ui.rawui.WindowSize = $uiproperty  
PS C:\Users> $host.UI.RawUI.WindowSize  
  
Width          Height  
---          ---  
 110           30
```

Similarly we may change the BufferSize property. This property is important since it determines the size of the history. The Height determines the number of lines of console history to be maintained. The Width property should be set to the same width as the WindowSize.

```
$uiproperty = $host.ui.rawui.BufferSize  
$uiproperty.Height = 1000  
$uiproperty.Width = 120  
$host.ui.rawui.BufferSize = $uiproperty
```

```
PS C:\Users> $host.UI.RawUI.BufferSize
Width----- Height-----
----- 120 ----- 3000

PS C:\Users> $uiproperty = $host.ui.rawui.BufferSize
PS C:\Users> $uiproperty.Height = 1000
PS C:\Users> $uiproperty.Width = 120
PS C:\Users>
PS C:\Users> $host.ui.rawui.BufferSize = $uiproperty
PS C:\Users>
PS C:\Users> $host.UI.RawUI.BufferSize
Width----- Height-----
----- 120 ----- 1000
```

Information for using PSGet and PSReadline to customize the console is found at the page linked below.

<http://blogs.technet.com/b/heyscriptingguy/archive/2014/06/16/the-search-for-a-better-powershell-console-experience.aspx>

Line Continuation

Tuesday, May 26, 2015
3:20 PM

PowerShell is interpretative so it will detect incomplete entries and prompt the user for more input using the *double-prompt* (">>"). In the example below, the closing " is missing causing the user to prompt for more input. PowerShell continues to prompt for input until the closing quote is entered.

```
PS C:\Users>
PS C:\Users> write-host "The closing double quote is missing
>> notice the double prompt. PowerShell is looking for the closing quote.
>> I didn't enter above so I'm prompted again. I'll enter the double-quote here"
>>
The closing double quote is missing
notice the double prompt. PowerShell is looking for the closing quote.
I didn't enter above so I'm prompted again. I'll enter the double-quote here
PS C:\Users>
```

After entering the double-quote to terminate the string, we still see the double-quote providing the opportunity for more input. Pressing enter twice informs PowerShell that no more input is coming. At that point, PowerShell immediately executes the write-host command. Sometimes the double-prompt mode is entered inadvertently. In this case, pressing <ctrl-c> aborts the current entry.

PowerShell many times guesses that additional keyboard input is required because the command entered is incomplete. In the example below, we attempt to do some simple arithmetic. Since nothing is entered after the division operator PowerShell prompts for more input. The equation is completed by entering the 2 on the double-prompt line. Again, the enter key is pressed twice to cause PowerShell to execute the operation and return a result.

```
PS C:\Users>
PS C:\Users> 2 + 2 /
>> 2
>>
3
```

In the next example, we want PowerShell to perform the calculation $(2 + 2) / 2$. Where the line breaks is important since PowerShell needs some clue as to whether additional input is coming. PowerShell looks at the first entry, $(2 + 2)$, as complete and provides the answer. There was nothing in that entry to indicate to PowerShell that additional input is required. The second set of entries remedies this problem. The missing closing parentheses causes PowerShell to prompt for more input. An alternative is illustrated in the third entry where the line continuation metacharacter (grave or back tick `) terminates the entry. This informs PowerShell that more input is required so the double-prompt appears. Allowing us to complete the equation.

```
PS C:\Users>
PS C:\Users> ( 2 + 2 )
4
PS C:\Users>
PS C:\Users> ( 2 + 2
>> ) / 2
>>
2
PS C:\Users>
PS C:\Users> ( 2 + 2 ) ` 
>> / 2
>>
2
```

The line continuation character is very important when writing scripts with cmdlets that have many parameters. This is best illustrated in the example below.

The first illustration shows the usual method for entering a cmdlet where the cmdlet parameters are entered on this same line. This becomes problematic when many parameters.

```
Untitled1.ps1* x
1 clear-host
2 write-host
3 Test-Connection -ComputerName 8.8.8.8 -BufferSize 100 -Count 1 -Delay 3 -TimeToLive 255
```

Source	Destination	IPV4Address	IPV6Address	Bytes
EL-CID	8.8.8.8	8.8.8.8		100

```
PS C:\Users>
```

In the next illustration, the script was altered to use line continuation to place the parameters on individual lines. This makes the script more readable and easier to modify.

```
Untitled1.ps1* X
1 clear-host
2 write-host
3 Test-Connection -ComputerName 8.8.8.8
4             -BufferSize 100
5             -Count 1
6             -Delay 3
7             -TimeToLive 255
```

```
<
-----
```

Source	Destination	IPV4Address	IPV6Address
EL-CID	8.8.8.8	8.8.8.8	

```
PS C:\Users>
```

Using DOS external commands

Saturday, January 12, 2013
9:06 PM

Generally there is no need to use the legacy external commands since a more powerful cmdlet is always available. However, using legacy commands during the transition to PowerShell ease that transition. In this section, we explore the use of the legacy commands in PowerShell.

The legacy netstat command returns information about current host connections. Listed below is typical netstat output.

Active Connections			
Proto	Local Address	Foreign Address	State
TCP	127.0.0.1:60774	127.0.0.1:65001	ESTABLISHED
TCP	127.0.0.1:65001	127.0.0.1:60774	ESTABLISHED
TCP	192.168.15.12:49241	157.56.98.50:443	ESTABLISHED
TCP	192.168.15.12:49382	172.225.9.83:443	CLOSE_WAIT
TCP	192.168.15.12:49420	192.168.15.6:8080	ESTABLISHED
TCP	192.168.15.12:50311	62.116.219.119:80	CLOSE_WAIT
TCP	192.168.15.12:50316	23.37.13.163:443	CLOSE_WAIT
TCP	192.168.15.12:51847	93.184.215.200:443	CLOSE_WAIT
TCP	192.168.15.12:51849	23.197.31.194:80	CLOSE_WAIT
TCP	192.168.15.12:51850	108.162.232.199:80	CLOSE_WAIT
TCP	192.168.15.12:51852	72.247.9.122:80	CLOSE_WAIT
TCP	192.168.15.12:51853	65.55.42.33:443	ESTABLISHED
TCP	192.168.15.12:51856	23.201.58.73:443	CLOSE_WAIT
TCP	192.168.15.12:52906	23.218.140.84:80	ESTABLISHED
TCP	192.168.15.12:52933	23.218.140.84:80	CLOSE_WAIT
TCP	192.168.15.12:52953	23.201.59.85:80	CLOSE_WAIT
TCP	192.168.15.12:52961	23.218.140.84:80	CLOSE_WAIT
TCP	192.168.15.12:52964	23.218.140.84:80	CLOSE_WAIT

The difference in using netstat in PowerShell is that the output from external commands may be saved in variable. A variable is just a location in the address space of the PowerShell session that can hold data.

The syntax below executes the netstat command and stores the output in the variable \$result. Piping \$result into get-member reveals that \$result is a string type (object). PowerShell implicitly *types* a variable based upon the data assigned to the variable. In this case, netstat outputs text data so \$results is typed as a String. Further, since netstat produces multiple lines of text, the variable \$result is an array of strings.

```
$result = netstat -n
$result | get-member
```

```

PS C:\Users>
PS C:\Users> $result = netstat -n
PS C:\Users> $result | get-member

```

TypeName: System.String

Name	MemberType	Definition
Clone	Method	System.Object Clone(), System.Object I
CompareTo	Method	int CompareTo(System.Object value), in
Contains	Method	bool Contains(string value)
CopyTo	Method	void CopyTo(int sourceIndex, char[] de
EndsWith	Method	bool EndsWith(string value), bool Ends
Equals	Method	bool Equals(System.Object obj), bool E
GetEnumerator	Method	System.CharEnumerator GetEnumerator()
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
GetTypeCode	Method	System.TypeCode GetTypeCode(), System.
IndexOf	Method	int IndexOf(char value), int IndexOf(c
IndexOfAny	Method	int IndexOfAny(char[] anyOf), int Inde
Insert	Method	string Insert(int startIndex, string v
IsNormalized	Method	bool IsNormalized(), bool IsNormalized
LastIndexOf	Method	int LastIndexOf(char value), int LastI
LastIndexOfAny	Method	int LastIndexOfAny(char[] anyOf), int
Normalize	Method	string Normalize(), string Normalize(S
PadLeft	Method	string PadLeft(int totalWidth), string
PadRight	Method	string PadRight(int totalWidth), string
Remove	Method	string Remove(int startIndex, int cour
Replace	Method	string Replace(char oldChar, char newC
Split	Method	string[] Split(Params char[] separator
StartsWith	Method	bool Startswith(string value), bool St
Substring	Method	string Substring(int startIndex), stri
ToBoolean	Method	bool IConvertible.ToBoolean(System.IFo
ToByte	Method	byte IConvertible.ToByte(System.IForma
ToChar	Method	char IConvertible.ToChar(System.IForma
ToCharArray	Method	char[] ToCharArray(), char[] ToCharArr
ToDateTime	Method	datetime IConvertible.ToDateTime(Syste
ToDecimal	Method	decimal IConvertible.ToDecimal(System.
ToDouble	Method	double IConvertible.ToDouble(System.IF
ToInt16	Method	int16 IConvertible.ToInt16(System.IFor
ToInt32	Method	int IConvertible.ToInt32(System.IForma
ToInt64	Method	long IConvertible.ToInt64(System.IFor
ToLower	Method	string ToLower(), string ToLower(cultu
ToLowerInvariant	Method	string ToLowerInvariant()
ToSByte	Method	sbyte IConvertible.ToSByte(System.IFor
ToSingle	Method	float IConvertible.ToSingle(System.IFo
ToString	Method	string ToString(), string ToString(Sys
ToType	Method	System.Object IConvertible.ToType(type
ToUInt16	Method	uint16 IConvertible.ToUInt16(System.IF
ToUInt32	Method	uint32 IConvertible.ToUInt32(System.IF
ToUInt64	Method	uint64 IConvertible.ToUInt64(System.IF
ToUpper	Method	string ToUpper(), string ToUpper(cultu
ToUpperInvariant	Method	string ToUpperInvariant()
Trim	Method	string Trim(Params char[] trimChars),
TrimEnd	Method	string TrimEnd(Params char[] trimChars
TrimStart	Method	string TrimStart(Params char[] trimCha
Chars	ParameterizedProperty	char Chars(int index) {get;}
Length	Property	int Length {get;}

We may use the property and methods of the String type to derive some useful information from \$result.

How many connections exist?

The variable \$result contains each line output by netstat, i.e., a *collection* of lines. To count the number of connections, we just need to count the number of lines since each connection is displayed on a separate line. The *length* property of the provides the number of elements in the array. Since each element holds a line of text, the value of length is the number of lines outputted by netstat.

```
$result.length
```

```
PS C:\Users>
PS C:\Users> $result.Length
137
```

\$result holds all of the output of netstat including any titles or column headings. Inspecting the contents of \$result shows the lines to exclude from the count.

```
PS C:\users>
PS C:\users> $result = netstat -n
PS C:\users>
PS C:\users> $result

Active Connections

Proto Local Address Foreign Address State
TCP   127.0.0.1:49276  127.0.0.1:65001 ESTABLISHED
TCP   127.0.0.1:65001  127.0.0.1:49276 ESTABLISHED
TCP   192.168.15.4:49238 157.56.100.146:443 ESTABLISHED
TCP   192.168.15.4:49297 74.125.69.188:5228 ESTABLISHED
```

We see the first four lines in \$result should not be counted; so we need to subtract 4 from \$result to find the actual number of connections.

```
$result.length - 4
```

What are the established connections?

Looking at the sample output of netstat above, we see that established connections have the keyword ESTABLISHED in the output. We just need to select those lines with that keyword or partial keyword. The Select-String cmdlet does just that. The Select-String cmdlet finds text in strings and files and outputs those lines that contain the match pattern. In the pipeline below each line in \$results is piped into select-string individually. The select-string cmdlet inspects the line looking for the substring "EST". If the substring is not found, the line is discarded. Any lines that match are output to the console.

PS C:\Users>	PS C:\Users> \$result select-string "EST"
TCP	127.0.0.1:60774
TCP	127.0.0.1:65001
TCP	192.168.15.12:49241
TCP	192.168.15.12:49420
TCP	192.168.15.12:51853
TCP	192.168.15.12:52906
TCP	192.168.15.12:53226
TCP	192.168.15.12:54581
TCP	192.168.15.12:54582
TCP	192.168.15.12:56971
TCP	192.168.15.12:57203
TCP	192.168.15.12:57204
TCP	192.168.15.12:57221
TCP	192.168.15.12:57226
TCP	192.168.15.12:57233
TCP	192.168.15.12:57263
TCP	192.168.15.12:57297
TCP	192.168.15.12:57298
TCP	192.168.15.12:57299
TCP	192.168.15.12:57439
TCP	192.168.15.12:57441
TCP	192.168.15.12:57461
TCP	192.168.15.12:57462
TCP	192.168.15.12:57463
TCP	192.168.15.12:57464
TCP	192.168.15.12:57481
TCP	192.168.15.12:57482
TCP	192.168.15.12:57483
	127.0.0.1:65001
	127.0.0.1:60774
	157.56.98.50:443
	192.168.15.6:8080
	65.55.42.33:443
	23.218.140.84:80
	65.55.42.33:443
	206.10.153.169:993
	173.194.192.109:993
	198.111.176.143:443
	216.239.120.246:80
	216.239.120.246:80
	64.30.236.32:80
	64.30.236.32:80
	63.158.227.32:80
	23.201.59.81:80
	63.158.227.65:80
	63.158.227.65:80
	63.158.227.65:80
	63.158.227.65:80
	63.158.227.65:80
	63.158.227.65:80
	65.126.84.153:80
	149.174.67.75:80
	64.30.234.60:2112
	64.30.235.231:443
	134.170.110.72:443
	134.170.110.72:443
	23.62.6.161:80
	23.201.88.229:443

Note, in this case the title and column headings are not a factor since they were filtered out by select-string.

What is the number of established connections?

First we select the lines with the that contain the substring "EST" as in the above. The output of select-string is saved in the variable \$est. From the explanation above for \$result, we know that \$est is a string type. The length property shows the number of matching lines which is essentially is the number of established connections.

```
$est = $result | select-string "EST"  
$est.length
```

```
PS C:\Users>  
PS C:\Users> $est = $result | select-string "EST"  
PS C:\Users> $est.length  
66
```

An alternative to creating another variable is to pipe the output of select-string to the measure-object cmdlet. The cmdlet measure-object provides some basic arithmetic functions one of which is count.

```
$result | select-string "EST" | measure-object
```

```
PS C:\Users>  
PS C:\Users> $result | select-string "EST" | measure-object  
  
Count      : 66  
Average    :  
Sum        :  
Maximum    :  
Minimum    :  
Property   :
```

The example below pipes the output of ipconfig directly into select-string. The select-string cmdlet uses the *regular expression* "ipv4|mask" to select any line containing the text ipv4 or the word mask.

```
ipconfig | select-string 'ipv4|mask'
```

```
PS C:\Users>
PS C:\Users> ipconfig | select-string 'ipv4|mask'

IPv4 Address. . . . . : 192.168.15.12
Subnet Mask . . . . . : 255.255.255.0
IPv4 Address. . . . . : 192.168.183.1
Subnet Mask . . . . . : 255.255.255.0
IPv4 Address. . . . . : 192.168.40.1
Subnet Mask . . . . . : 255.255.255.0
```

Sometimes the legacy command name conflicts with a PowerShell name. In the example below, the attempt is made to execute the "Service Control" executable sc.exe. The problem is that sc is the alias for the cmdlet set-content. This may be remedied by executing an instance of the cmd shell and running sc.exe within it.

```
PS C:\Users>
PS C:\Users> sc query

cmdlet Set-Content at command pipeline position 1
Supply values for the following parameters:
Value[0]: PS C:\Users>
PS C:\Users>
PS C:\Users> cmd /c sc query

SERVICE_NAME: AdobeARMservice
DISPLAY_NAME: Adobe Acrobat Update Service
    TYPE               : 10  WIN32_OWN_PROCESS
    STATE              : 4   RUNNING
                           (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
    WIN32_EXIT_CODE    : 0   (0x0)
    SERVICE_EXIT_CODE : 0   (0x0)
    CHECKPOINT        : 0x0
    WAIT_HINT         : 0x0

SERVICE_NAME: Appinfo
DISPLAY_NAME: Application Information
    TYPE               : 20  WIN32_SHARE_PROCESS
    STATE              : 4   RUNNING
                           (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
    WIN32_EXIT_CODE    : 0   (0x0)
    SERVICE_EXIT_CODE : 0   (0x0)
    CHECKPOINT        : 0x0
    WAIT_HINT         : 0x0
```

The output returned by the cmd shell also be piped as text into a cmdlet as shown below. In this example, the sc command lists information about services. Since sc provides a significant amount of output, we pipe that output to select-string to select only the lines containing the display name of the service.

```
cmd /c sc query | select-string 'DISPLAY_NAME'
```

```
PS C:\Users>
PS C:\Users> cmd /c sc query | select-string 'DISPLAY_NAME'

DISPLAY_NAME: Adobe Acrobat Update Service
DISPLAY_NAME: Application Information
DISPLAY_NAME: Windows Audio Endpoint Builder
DISPLAY_NAME: Windows Audio
DISPLAY_NAME: Base Filtering Engine
DISPLAY_NAME: Background Intelligent Transfer Service
DISPLAY_NAME: Background Tasks Infrastructure Service
DISPLAY_NAME: Computer Browser
DISPLAY_NAME: Skype Click to Call Updater
DISPLAY_NAME: Skype Click to Call PNR Service
DISPLAY_NAME: Cryptographic Services
DISPLAY_NAME: Offline Files
DISPLAY_NAME: Sound Blaster Audio Service
DISPLAY_NAME: DCOM Server Process Launcher
DISPLAY_NAME: Device Association Service
DISPLAY_NAME: DHCP Client
DISPLAY_NAME: Diagnostics Tracking Service
```

The legacy commands are rarely used since PowerShell implements cmdlets that perform many of the same functions.

Transcripts

Saturday, December 28, 2013
3:16 PM

One of the many weaknesses of the batch files running in the legacy command shell is the inability to record the execution of a command and its subsequent output. This was particularly problematic when the batch file was run in the background using the task scheduler.

The developers of PowerShell addressed this shortcoming with *transcripts*. A transcript is a record of the PowerShell session from the time the transcript is started to the time the transcript is terminated or the console closed. The transcript is a text file that contains every PowerShell command executed along with its output and errors. Legacy external commands such as ping and ipconfig will have the command recorded but not the output.

The `Start-Transcript` cmdlet creates a record of all or part of a Windows PowerShell session in a text file. The `Stop-Transcript` cmdlet stops transcription.

The illustration below demonstrates the use of transcripts to capture a session.

```
PS C:\users>
PS C:\users> start-transcript myps.txt
Transcript started, output file is myps.txt
PS C:\users> get-childitem

Directory: C:\users

Mode          LastWriteTime    Length Name
----          -----        ----- 
d---          3/13/2015     3:26 PM   mgalea
d-r--          7/14/2009     3:45 AM   Public
-a---          5/8/2015      3:59 PM   530 myps.txt
-a---          5/8/2015      3:06 PM   435 psex1.ps1
-a---          5/8/2015      3:02 PM   419 Untitled1.ps1
-a---          5/8/2015      3:06 PM   24 z.txt

PS C:\users> Push-Location
PS C:\users> set-location c:\windows
PS C:\windows> pop-location
PS C:\users> stop-transcript
Transcript stopped, output file is C:\users\myps.txt
PS C:\users>
```

We use the `get-content` cmdlet to list the contents of the transcript file.

```

PS C:\users>
PS C:\users> get-content .\myps.txt
*****
Windows PowerShell transcript start
Start time: 20150508155944
Username : mink\mgalea
Machine   : MINK (Microsoft Windows NT 6.1.7601 Service Pack 1)
*****
Transcript started, output file is myps.txt
PS C:\users> get-childitem

    Directory: C:\users

Mode                LastWriteTime       Length Name
----                -----          ----
d----        3/13/2015      3:26 PM           mgalea
d-r--        7/14/2009      3:45 AM           Public
-a---        5/8/2015       3:59 PM         530 myps.txt
-a---        5/8/2015       3:06 PM         435 psex1.ps1
-a---        5/8/2015       3:02 PM         419 Untitled1.ps1
-a---        5/8/2015       3:06 PM          24 z.txt

PS C:\users> Push-Location
PS C:\users> set-location c:\windows
PS C:\windows> pop-location
PS C:\users> stop-transcript
*****
Windows PowerShell transcript end
End time: 20150508160029
*****
PS C:\users>

```

* PowerShell Profile Script

Thursday, June 25, 2015
11:25 AM

This page is reserved for future content.

The ISE

Saturday, December 28, 2013
2:37 PM

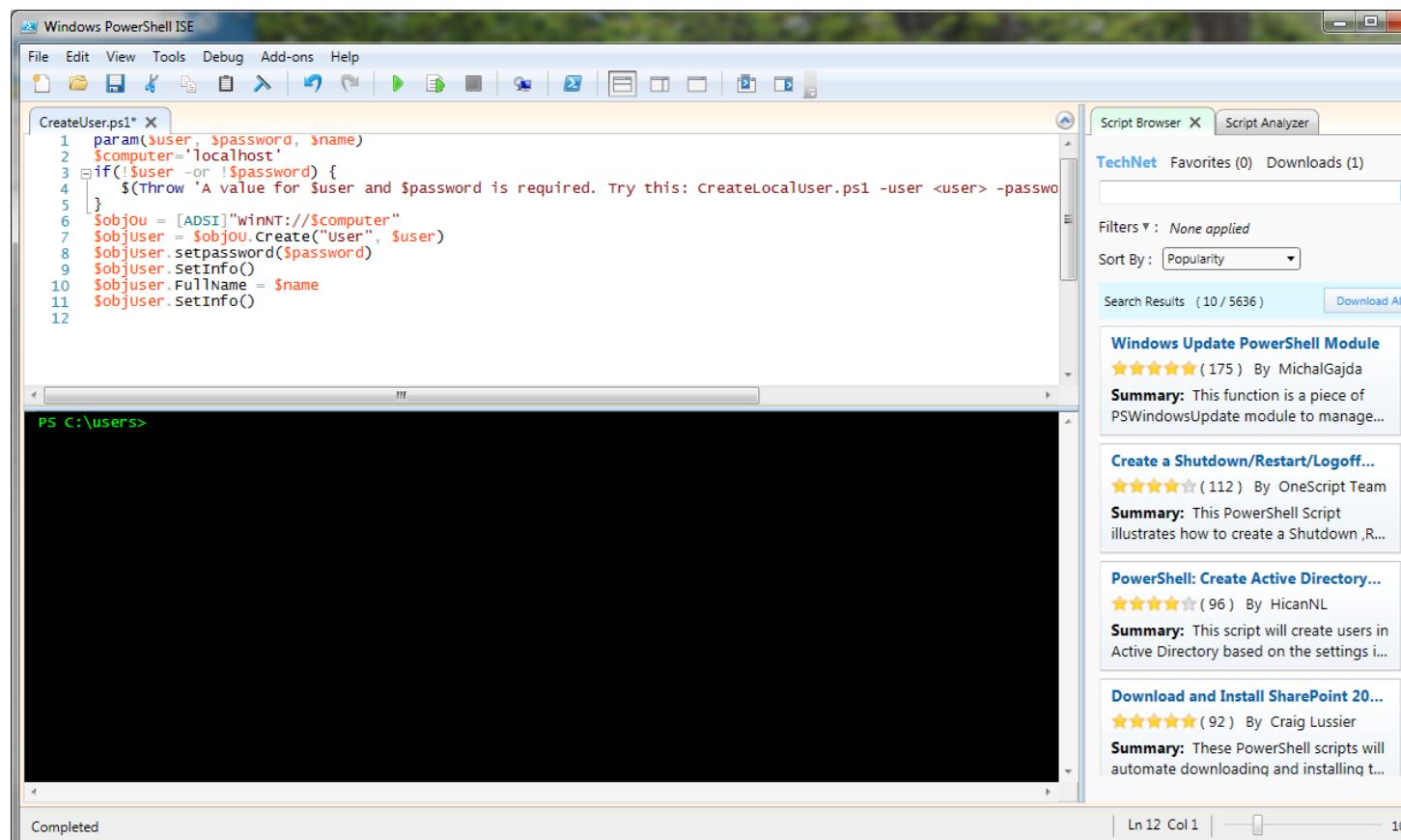
The PowerShell Integrated Scripting Environment (ISE) is used to develop and debug scripts

The PowerShell console executable, *powershell_ise.exe*, is found in
C:\windows\system32\WindowsPowerShell\v1.0

The ISE supports multiple tabs where each tab may contain a different script. Each tab is also a different *runspace*. A runspace is a separate instance of PowerShell.

The ISE consists of two panes: the script pane and the console pane. The script pane contains the PowerShell script and supports multiple tabs. The console pane shows the output of the script and is also used during debugging.

The ISE is not available on Server Core 2012 or Hyper-V Server 2012 since these operating systems do not support a graphical user environment.



The screenshot shows the Windows PowerShell ISE interface. The main window has a title bar "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. The left pane is a script editor containing a file named "CreateUser.ps1" with the following code:

```
1 param($user, $password, $name)
2 $computer='localhost'
3 if($user -or !$password) {
4     $throw 'A value for $user and $password is required. Try this: createLocalUser.ps1 -user <user> -passwo
5 }
6 $objou = [ADSI]"winNT://$computer"
7 $objuser = $objou.Create("User", $user)
8 $objuser.SetPassword($password)
9 $objuser.SetInfo()
10 $objuser.FullName = $name
11 $objuser.SetInfo()
```

The right pane is a "Script Browser" window showing a list of PowerShell scripts from TechNet. The first item listed is "Windows Update PowerShell Module". Other items include "Create a Shutdown/Restart/Logoff...", "PowerShell: Create Active Directory...", and "Download and Install SharePoint 20...".

More information on using the ISE is available at [Using the PowerShell ISE](#)

Media

Saturday, August 8, 2015
8:03 AM

Windows Powershell Tutorial - Get-Alias, Set-Alias

https://www.youtube.com/watch?v=uVqB_WFaJm8

Creating a New Alias and adding it

https://www.youtube.com/watch?v=1_JvPQX2H4E

Aliases

Monday, July 21, 2014
2:54 PM

Aliases are another name for a cmdlet for executable. Aliases allow us to create shorter more easily remembered names for cmdlets or executables that we use frequently. Aliases have been a part of shells and scripting languages since the very beginning. Aliases are useful in associating a shortcut, or abbreviated name, with a cmdlet or with a snippet of PowerShell code. PowerShell has a number of intrinsic aliases. As users, we may create aliases once the session is established. User-created aliases do not persist between sessions. Persisting user-created aliases requires that the cmdlet be added to the PowerShell *profile script*.

Get-Alias:

This cmdlet lists the contents of the alias: drive. Parameters -name and -definition allow selection criteria to be provided.

```
# list all aliases
get-alias

# list the alias for the specific command defined
get-alias -definition import-alias

# list the alias for the commands defined
get-alias -definition import-alias, import-csv

# list aliases whose name begins with g but exclude anything that
begins with Group
get-alias -name g* -exclude Group*

# list the aliases for all cmdlets whose name begins with import
get-alias -definition import*

# an alternative easy to select aliases
get-alias | where-object { $_.definition -eq 'get-process' }
```

New-Alias:

As the name implies, the new-alias cmdlet creates a new alias in the current PowerShell session. The alias is not save once the session ends.

In the example below, we create an alias called "lst" associated with get-childitem. This alias cannot be deleted or overwritten during the session as specified by -option readonly. Note the use of the backtick ` character to allow multi-line input.

```
New-Alias -Name lst -Value get-childitem `
           -Description 'shortcut for get-childitem' `
           -Option readonly
```

```

PS C:\user> New-Alias -Name lst -Value get-childitem `.
>>           -Description 'shortcut for get-childitem'
>>           -Option readonly
>>
PS C:\user> get-alias | where-object { $_.Name -eq 'lst' }

 CommandType      Name                               ModuleName
 -----          ----
 Alias            lst -> Get-ChildItem

PS C:\user> get-alias | where-object { $_.Name -eq 'lst' } | format-table name, descrip
Name        Description
----        -----
lst         shortcut for get-childitem

```

In the example below, we create a new alias *ed* which is an alias for notepad.exe. After the command executes, we may execute notepad by typing *ed* on the console.

```
new-alias -name 'ed' -value 'notepad.exe'
```

```

PS C:\user>
PS C:\user> new-alias -name 'ed' -value 'notepad.exe'
PS C:\user>
PS C:\user> get-alias | where-object { $_.Definition -like 'note*' }

 CommandType      Name                               ModuleName
 -----          ----
 Alias            ed -> notepad.exe

```

Set-Alias:

This set-alias cmdlet either modifies and existing alias or creates a new one. When the cmdlet is executed, if the alias already exists in the alias: drive, the specified alias is altered using the information provided to set-alias. If the alias does not exist in the alias: drive, a new alias is created.

In this example, we create or overwrite the alias "pingit". The -force switch allows overwriting an existing alias with the same name.

```
Set-Alias -Name pingit -Value test-connection `.
           -Description 'Pings a host using test-connection' `
```

```
-Force
```

```
PS C:\user> Set-Alias -Name pingit -Value test-connection ` 
>>             -Description 'Pings a host using test-connection' ` 
>>             -Force
>>
PS C:\user> get-alias | where-object { $_.Name -like 'ping*' } | format-table name, description -auto
Name      Description
----      -----
pingit   Pings a host using test-connection
```

```
set-alias -name 'ed' -value 'C:\Program Files
(x86)\Notepad++\notepad++.exe'
```

```
PS C:\user>
PS C:\user> get-alias | where-object { $_.Definition -like '*note*' }
 CommandType      Name                               ModuleName
-----      ----
 Alias          ed -> notepad.exe

PS C:\user> set-alias -name 'ed' -value 'C:\Program Files (x86)\Notepad++\notepad++'
PS C:\user>
PS C:\user> get-alias | where-object { $_.Definition -like '*note*' }
 CommandType      Name                               ModuleName
-----      ----
 Alias          ed -> notepad++.exe
```

In the next example, we use set-alias to create an alias called out-clipboard that may be used in a pipeline to direct output to the clipboard. To reference the location of the clip.exe, the environment variable SystemRoot is used. In most Windows systems, the contents of this variable is c:\windows. The little-known executable clip.exe accepts text input and copies it to the clipboard.

```
new-alias Out-Clipboard $env:SystemRoot\system32\clip.exe
```

```

PS C:\user>
PS C:\user> new-alias Out-Clipboard $env:SystemRoot\system32\clip.exe
PS C:\user> get-alias | where-object { $_.Definition -like '*clip*' }

 CommandType      Name
 ----          ----
 Alias           Out-Clipboard

```

Having created the alias, we may used it at the end of a pipeline to copy text to the clipboard. In the example, the output of test-connection (ping) is piped to clip.exe which copies the text to the clipboard. The contents of the clipboard may be used with any application that allows pasting from the clipboard.

```

PS C:\user>
PS C:\user> test-connection www.google.com | Out-Clipboard
PS C:\user>

```

Export-Alias:

As the name implies, this cmdlet exports aliases to a text file. If the -name parameter is not specified all aliases including the built-in PowerShell aliases are exported.

```

# export all aliases to the file allalias.txt
Export-Alias allalias.txt

# export the alias pingit and l to the file myalias.txt
Export-Alias -Name pingit, lst -path myalias.txt

```

The myalias.txt file contains the following:

```

# Alias File
# Exported by : CIS
# Date/Time : Monday, July 21, 2014 6:02:05 PM
# Computer : TI24401-16577
"pingit","test-connection","Pings a host using test-
connection","None"
"lst","get-childitem","shortcut for get-childitem","ReadOnly"

```

Import-Alias:

As the name indicates import-alias imports a text file in the form created by export-alias into the current session. Since aliases are not preserved once the session is closed, the aliases could be exported before closing the PowerShell session then importing the aliases when a new session is opened. A better approach would be to create a PowerShell profile scripts that automatically imports that aliases when the PowerShell console is opened.

```
Import-Alias -path myalias.txt
```

The Alias Drive

Monday, December 30, 2013
11:02 AM

The alias: drive holds all the aliases built-in in PowerShell and any aliases created by the user during the session.

```
get-childitem alias: | out-host -paging
```

```
PS C:\users>
PS C:\users> get-childitem alias: | out-host -paging
```

CommandType	Name	ModuleNam
Alias	% -> ForEach-Object	
Alias	? -> Where-Object	
Alias	ac -> Add-Content	
Alias	asnp -> Add-PSSnapin	
Alias	cat -> Get-Content	
Alias	cd -> Set-Location	
Alias	chdir -> Set-Location	
Alias	clc -> Clear-Content	
Alias	clear -> Clear-Host	
Alias	clhy -> Clear-History	
Alias	cli -> Clear-Item	
Alias	clp -> Clear-ItemProperty	
Alias	cls -> Clear-Host	
Alias	clv -> Clear-Variable	
Alias	cnsn -> Connect-PSSession	
Alias	compare -> Compare-Object	
Alias	copy -> Copy-Item	

The above display may be misleading since the name column shows both the alias name and definition. Before going further let's look at the properties of the alias object.

```
get-childitem alias: | get-member
```

```

PS C:\users>
PS C:\users> get-childitem alias: | get-member

TypeName: System.Management.Automation.AliasInfo

Name           MemberType
----           -----
Equals         Method
GetHashCode   Method
GetType        Method
ResolveParameter Method
ToString       Method
PSDrive        NoteProperty
PSIsContainer  NoteProperty
PSPath         NoteProperty
PSProvider     NoteProperty
CommandType    Property
Definition    Property
Description   Property
Module         Property
ModuleName    Property
Name          Property
Options        Property
OutputType     Property
Parameters     Property
ParameterSets  Property
ReferencedCommand Property
RemotingCapability Property

Name           Definition
----           -----
bool Equals(System.Object obj)
int GetHashCode()
type GetType()
System.Management.Automation.ParameterMetadata
string ToString()
System.Management.Automation.PSDriveInfo PSDrive
System.Boolean PSIsContainer=False
System.String PSPath=Microsoft.PowerShell.Commands.PSPath
System.Management.Automation.ProviderInfo PSProvider
System.Management.Automation.CommandTypes CommandType
string Definition {get;}
string Description {get;set;}
psmoduleinfo Module {get;}
string ModuleName {get;}
string Name {get;}
System.Management.Automation.ScopedItemOption Options
System.Collections.ObjectModel.ReadOnlyCollection`1 OutputType
System.Collections.Generic.Dictionary`2 Parameters
System.Collections.ObjectModel.ReadOnlyCollection`1 ParameterSets
System.Management.Automation.CommandInfo ReferencedCommand
System.Management.Automation.RemotingCapability RemotingCapability

```

From the properties we see in the above, let's look at the name and definition separately.

```
get-childitem alias: | select-object name, definition | format-table  
-autosize
```

```

PS C:\users>
PS C:\users> get-childitem alias: | select-object name, definition | format-table -auto

```

Name	Definition
%	ForEach-Object
?	Where-Object
ac	Add-Content
asnp	Add-PSSnapIn
cat	Get-Content
cd	Set-Location
chdir	Set-Location
clc	Clear-Content
clear	Clear-Host
cchy	Clear-History
cli	Clear-Item
clip	Clear-ItemProperty
cls	Clear-Host
clv	Clear-Variable
cnsn	Connect-PSSession
compare	Compare-Object
copy	Copy-Item
cp	Copy-Item
cpi	Copy-Item
cpp	Copy-ItemProperty
curl	Invoke-WebRequest
cvpa	Convert-Path
dbp	Disable-PSBreakpoint
rm	Remove-Item

From the display we see that name is the alias name and the definition is the name of the cmdlet to which the alias corresponds. Using this information, we may retrieve the alias of any cmdlet using the pipelines below. We just need to alter the name or pattern to match the cmdlet for which we are searching. In the examples below we are searching for the alias for get-process.

```

# The -eq operator attempts and exact match on the pattern ('get-process')
get-childitem alias: | where-object { $_.definition -eq 'get-process' }

# When the -like operator is used wildcards and a partial match
# string may be used
get-childitem alias: | where-object { $_.definition -like '*process*' }

```

```
PS C:\user> get-childitem alias: | where-object { $_.definition -eq 'get-process' }

 CommandType      Name                               ModuleName
 -----          ----
 Alias           gps -> Get-Process
 Alias           ps  -> Get-Process

PS C:\user> get-childitem alias: | where-object { $_.definition -like '*process*' }

 CommandType      Name                               ModuleName
 -----          ----
 Alias           gps -> Get-Process
 Alias           kill -> Stop-Process
 Alias           ps  -> Get-Process
 Alias           saps -> Start-Process
 Alias           spps -> Stop-Process
 Alias           start -> Start-Process
```

Media

Saturday, August 8, 2015
8:05 AM

PowerShell - Working with Variables

<https://www.youtube.com/watch?v=2eiNFuBjTnc>

Windows Power Shell Tutorial part 2 PowerShell Variables

<https://www.youtube.com/watch?v=rrUvB7rPkts>

PowerShell Basics: Using variables - Part 1

<https://www.youtube.com/watch?v=ZwQR0ljeHqI>

Variables

Tuesday, January 15, 2013
7:16 AM

Variables are a means of storing results of operations so they may be referenced later. A variable may be *scalar* or an *array*. A scalar variable holds one object. An array holds multiple objects. Arrays are sometimes called *collections*.

We use the assignment operator "=" to assign values to variables as below:

```
$var = 10
```

The above statement is read as "\$var is assigned the value 10". In the illustration below the GetType method, which is available in all objects, shows that the type (name) is Int32 (32 bit integer) and that the BaseType is a ValueType (scalar).

```
$var = 10
$var.GetType()
```

```
PS C:\user>
PS C:\user> $var = 10
PS C:\user> $var.GetType()

IsPublic IsSerial Name                                     BaseType
-----  -----  --  -----
True      True    Int32                                System.ValueType
```

The assignment statement in the illustration below shows one way in which an array is created. Assigning a variable a comma-separated list of value cause PowerShell to change the type of the variable to an array.

Using the GetType method confirmst that the BaseType of \$ary is Array. Albeit, the we assigned the variable a list on integers (see above), the type (name) is Object[] and not Int32. Since PowerShell arrays may hold objects of different

types (integers, string,...) , a specific type cannot be assigned so it is just classified as an object. The syntax Object[] means any element in the \$ary array may also be an array.

The Length property shows size of the array, i.e., the number of elements, which is 6. All collections in PowerShell are *zero indexed* which means that the first element is element 0. The illustration below also shows how to access the first ([0]), fourth ([3]), and sixth ([5]) elements of the array.

```
$ary = 1,2,3,10,120,500  
$ary.GetType()
```

```
PS C:\user>  
PS C:\user> $ary = 1,2,3,10,120,500  
PS C:\user> $ary.GetType()  
  
IsPublic IsSerial Name                                     BaseType  
-----  -----  ----  
True     True    Object[]  
  
PS C:\user> $ary.Length  
6  
PS C:\user> $ary[0]  
1  
PS C:\user> $ary[3]  
10  
PS C:\user> $ary[5]  
500
```

Depending on the cmdlet used and how it's used, the result may be a scalar or an array. An example of this behavior is the cmdlet test-connection.

```
$pingresult = test-connection 207.75.134.1  
$pingresult.GetType()
```

```
PS C:\Users>
PS C:\Users> $pingresult = test-connection 207.75.134.1
PS C:\Users> $pingresult.GetType()

IsPublic IsSerial Name                                     BaseType
----- -----   ----
True      True    Object[]                               System.Array

PS C:\Users> $pingresult.length
4
PS C:\Users>
```

The reason that `test-connection` returned an array of `PingStatus` objects is that by default, just like `ping`, `test-connection` pings the host four times. Using the `count` parameter with a value of one cause `test-connection` to ping the host just one time results in a scalar variable.

```
$pingresult = test-connection 207.75.134.1 -count 1
$pingresult.GetType()
```

```
PS C:\Users>
PS C:\Users> $pingresult = test-connection 207.75.134.1 -count 1
PS C:\Users>
PS C:\Users> $pingresult.GetType()

IsPublic IsSerial Name                                     BaseType
----- -----   ----
True      True    ManagementObject                  System.Managem...
```



```
PS C:\Users> $pingresult.length
PS C:\Users> -
```

Note that in the above the `length` property of the variable `$pingresult` is null, *no value*. Scalar variables have no length so their `length` property will always be null.

The *type* of a variable determines the properties and methods that the variable possesses. A variable may be typed *implicitly*, by how the variable is used, or *explicitly* by how it is declared.

Programming languages are classified either as *strongly-typed* or *weakly-typed*. Strongly-typed languages require the explicit declaration of the type of a variable before it can be used. Weakly-typed languages support implicit typing and, in

some cases, do not provide for explicit typing. PowerShell provides both implicit and explicit typing of a variable.

In the illustration below, PowerShell automatically makes \$myint an integer based upon the assignment of the integer value 1024. The second example again shows implicit typing. The variable \$hisint is *cast* as a double precision floating point type; floating point types are typically used in scientific calculation where very large or very small numbers need to be represented. The final example demonstrates explicit typing of the variable \$hisint. In this case, PowerShell *recasts* the variable from Double to Int32. The variable \$hisint is typed as an integer regardless that the value assigned. The value 2048.25 would normally be interpreted by PowerShell as a floating point number. However due to the explicit [int] typing of \$histint, PowerShell truncated the number before assigning it to the variable.

```
$myint = 1024
$myint.GetType()

$hisint = 2048.25
$hisint.GetType()

[int] $hisint = 2048.25
$hisint.GetType()
```

```

PS C:\user>
PS C:\user> $myint = 1024
PS C:\user> $myint.GetType()

IsPublic IsSerial Name                                     BaseType
----- ----- ----
True      True     Int32                                System.ValueType

PS C:\user>
PS C:\user> $hisint = 2048.25
PS C:\user> $hisint.GetType()

IsPublic IsSerial Name                                     BaseType
----- ----- ----
True      True     Double                               System.ValueType

PS C:\user>
PS C:\user> [int] $hisint = 2048.25
PS C:\user> $hisint.GetType()

IsPublic IsSerial Name                                     BaseType
----- ----- ----
True      True     Int32                               System.ValueType

```

Simple PowerShell variable types are listed below.

[string]	Fixed-length string of Unicode characters
[char]	A Unicode 16-bit character
[byte]	An 8-bit unsigned character
[int]	32-bit signed integer
[long]	64-bit signed integer
[bool]	Boolean True/False value
[decimal]	A 128-bit decimal value
[single]	Single-precision 32-bit floating point number
[double]	Double-precision 64-bit floating point number
[DateTime]	Date and Time
[xml]	Xml object
[array]	An array of values
[hashtable]	Hashtable object

There are also hundreds of .Net types which will not be discussed.

The illustration below is an example of the disadvantage of explicit typing. \$lvar is explicit typed as a 32 bit integer. When the second assignment statement is attempted, PowerShell throws an error since the integer value of 10GB (10737418240) cannot fit into a 32 bit integer.

```
[int] $lvar = 1GB  
$lvar  
$lvar.GetType()  
$lvar = 10GB
```

```
PS C:\Users>  
PS C:\Users> [int] $lvar = 1GB  
PS C:\Users> $lvar  
1073741824  
PS C:\Users> $lvar.GetType()  


| IsPublic | IsSerial | Name  | BaseType         |
|----------|----------|-------|------------------|
| True     | True     | Int32 | System.ValueType |

  
PS C:\Users> $lvar = 10gb  
Cannot convert value "10737418240" to type "System.Int32". Error: "Value was outside the bounds of a System.Int32."  
At line:1 char:1  
+ $lvar = 10gb  
+ ~~~~~~  
+ CategoryInfo : MetadataError: (:) [], ArgumentTransformationMetadataException  
+ FullyQualifiedErrorId : RuntimeException  
  
PS C:\Users>
```

Once a variable is explicitly typed, PowerShell will not recast it to another type as evidenced by the error above. For an explicitly type variable to be recast to another type, it must be explicitly typed to the new type.

In the illustration below the same sequence of instructions are executed with the exception that \$lvar is not explicitly typed.

```

PS C:\Users>
PS C:\Users> $lvar = 1GB
PS C:\Users> $lvar
1073741824
PS C:\Users> $lvar.GetType()

IsPublic IsSerial Name                                     BaseType
-----  -----   ----
True      True    Int32                                System.ValueType

PS C:\Users> $lvar = 10gb
PS C:\Users> $lvar
10737418240
PS C:\Users> $lvar.GetType()

IsPublic IsSerial Name                                     BaseType
-----  -----   ----
True      True    Int64                                System.ValueType

```

What is demonstrated in the above is the advantage of not explicitly typing. In the second assignment statement PowerShell automatically *recasts* the int32 ([int]) to an int64 ([long]).

Built-in Variables:

There are a number of built-in variables in PowerShell. The ones most commonly encountered are \$true, \$false, \$null which are used many times for comparison. As the name implies \$true and \$false represent their corresponding boolean values. \$null represents the absence of a value.

All variables used in PowerShell are initialized with a value of \$null. Once a variable has been assigned a value, it may never have a null value unless explicitly assigned the value of \$null.

In the assignment statement \$str = "", the literal value "", which is represented by two successive single quotes or double quotes, is called the *null string*. A null string has a length of zero. However, as evidenced below, a null string is not null. It simply is a string of zero length.

```
PS C:\Users>
PS C:\Users> $var -eq $null
True
PS C:\Users> $var = 0
PS C:\Users>
PS C:\Users> $var -eq $null
False
PS C:\Users> $var = $null
PS C:\Users>
PS C:\Users> $var -eq $null
True
PS C:\Users>
PS C:\Users> $str -eq $null
False
PS C:\Users> $str = 'abc'
PS C:\Users>
PS C:\Users> $str -eq $null
False
PS C:\Users> $str = '' # assign the null string
PS C:\Users>
PS C:\Users> $str.length
0
PS C:\Users>
PS C:\Users> $str -eq $null
False
PS C:\Users>
```

Populating Variables:

The syntax of scripting languages has historically been terse in the effort to make their execution more efficient. PowerShell has adopted this philosophy as demonstrated below.

The traditional way of populating multiple variables with the same value requires multiples assignment statements:

```
$a = 10
$b = 10
$c = 10
```

A more shortened syntax is supported by PowerShell (and most programming languages):

```
$a = $b = $c = 10
```

Shown below is the traditional way of populating multiple variables with different values:

```
$a = 10  
$b = 20  
$c = 30
```

PowerShell supports a more unorthodox syntax as shown below. However, this could make the script less readable so be careful.

```
$a, $b, $c = 10, 20, 30
```

Exchanging the contents of two variables is a very common scripting/programming practice. The traditional method requires a third variable as shown below.

Given:

```
$a = 10  
$b = 20
```

We want to exchange the contents of \$a and \$b. Typically this requires a third variable \$c as shown:

```
$c = $a  
$a = $b  
$b = $c
```

PowerShell provides syntax that makes this exchange possible without a third variable as shown below. Obviously PowerShell Internally allocates a third variable to make this exchange. The only relief is that syntactically, we don't need to specify the third variable.

```
$a, $b = $b, $a
```

.Net Types

The above are simple variable types. We may also create variables of more complex types using the .Net classes. For sake of organization .Net classes are organized into *namespaces*. In PowerShell the syntax for defining a .Net class is `System.Net.<namespace>.<class>`. For example, .Net has a `NetworkInformation` namespace. That namespaces contains a class called `Ping`. The syntax for fully describing this class is `System.Net.NetworkInformation.Ping`. The example below shows how to initialize a PowerShell variable by instantiating an object from the *ping class (type)*. T

```
$pinger = new-object System.Net.NetworkInformation.Ping  
$pinger | gm
```

```
PS C:\Users>  
PS C:\Users> $pinger = new-object System.Net.NetworkInformation.Ping  
PS C:\Users> $pinger | gm
```

Name	MemberType	Definition
Disposed	Event	<code>System.EventHandler Disposed(System.Object sender, System.EventArgs e)</code>
PingCompleted	Event	<code>System.Net.NetworkInformation.PingCompletedEventHandler PingCompleted</code>
CreateObjRef	Method	<code>System.Runtime.Remoting.ObjRef CreateObjRef(string type)</code>
Dispose	Method	<code>void Dispose(), void IDisposable.Dispose()</code>
Equals	Method	<code>bool Equals(System.Object obj)</code>
GetHashCode	Method	<code>int GetHashCode()</code>
GetLifetimeService	Method	<code>System.Object GetLifetimeService()</code>
GetType	Method	<code>type GetType()</code>
InitializeLifetimeService	Method	<code>System.Object InitializeLifetimeService()</code>
Send	Method	<code>System.Net.NetworkInformation.PingResult Send(string hostNameOrAddress, int timeout)</code>
SendAsync	Method	<code>void SendAsync(string hostNameOrAddress, int timeout, System.Threading.Tasks.TaskCompletionSource<System.Net.NetworkInformation.PingResult> result)</code>
SendAsyncCancel	Method	<code>void SendAsyncCancel()</code>
SendPingAsync	Method	<code>System.Threading.Tasks.Task<System.Net.NetworkInformation.PingResult> SendPingAsync(string hostNameOrAddress, int timeout)</code>
ToString	Method	<code>string ToString()</code>
Container	Property	<code>System.ComponentModel.IContainer Container {get;}</code>
Site	Property	<code>System.ComponentModel.ISite Site {get; set;}</code>

The `$pinger` method `Send` may now be used to ping hosts.

```

PS C:\Users>
PS C:\Users> $pinger.send('www.wccnet.edu')

Status      : Success
Address    : 198.111.176.8
RoundtripTime : 0
Options     : System.Net.NetworkInformation.PingOptions
Buffer       : {97, 98, 99, 100...}

```

Another example is the NetworkInterface class found in the same namespace. To instantiate all of the network interface objects the method GetAllNetworkInterfaces must be used.

```

$adapters =
[System.Net.NetworkInformation.NetworkInterface]::GetAllNetworkInterfaces()
$adapters | gm

```

```

PS C:\user>
PS C:\user> $adapters = [System.Net.NetworkInformation.NetworkInterface]::GetAllNetworkInterfaces()
PS C:\user> $adapters | gm

TypeName: System.Net.NetworkInformation.SystemNetworkInterface

Name          MemberType  Definition
----          -----  -----
Equals        Method     bool Equals(System.Object obj)
GetHashCode   Method     int GetHashCode()
GetIPProperties Method    System.Net.NetworkInformation.IPInterfaceProperties GetIPProperties()
GetIPStatistics Method   System.Net.NetworkInformation.IPInterfaceStatistics GetIPStatistics()
GetIPv4Statistics Method  System.Net.NetworkInformation.IPv4InterfaceStatistics GetIPv4Statistics()
GetPhysicalAddress Method System.Net.NetworkInformation.PhysicalAddress GetPhysicalAddress()
GetType       Method     type GetType()
Supports     Method    bool Supports(System.Net.NetworkInformation.NetworkInterfaceComponent)
ToString      Method    string ToString()
Description   Property   string Description {get;}
Id           Property   string Id {get;}
IsReceiveOnly Property  bool IsReceiveOnly {get;}
Name         Property   string Name {get;}
NetworkInterfaceType Property System.Net.NetworkInformation.NetworkInterfaceType NetworkInterfaceType
OperationalStatus  Property System.Net.NetworkInformation.OperationalStatus OperationalStatus
Speed         Property   long Speed {get;}
SupportsMulticast Property bool SupportsMulticast {get;}

```

Because multiple network interfaces exist, multiple interface objects are returned causing the \$adapter to be cast as an array. Piping the variable into get-member exposes the properties and methods of the SystemNetworkInterface object. We observe a couple of interesting methods for this object: GetIPProperties and GetPhysicalAddress.

The first entry in the array \$adapters[0] provides information about first network adapter. Piping \$adapters[0] into format-list provides a list of all the properties. We next use the GetIpProperties method to show the IP information for this adapter. Absent in the display is the information about the address assigned to the adapter. However, we see a nest object called UnicastAddresses which should provide the information in question. We execute the method using the syntax \$adapters[0].GetIpProperties().UnicastAddresses to display the addressing information.

```
$adapters[0] | fl *
$adapters[0].GetIpProperties()
$adapters[0].GetPhysicalAddress()
$adapters[0].GetIpProperties().UnicastAddresses
```

```
PS C:\user>
PS C:\user> $adapters[0] | fl *

Id : {6C4B7763-8C13-4C0C-B5EB-44C194436272}
Name : Local Area Connection
Description : Broadcom NetXtreme 57xx Gigabit Controller
NetworkInterfaceType : Ethernet
OperationalStatus : Up
Speed : 10000000000
IsReceiveOnly : False
SupportsMulticast : True

PS C:\user> $adapters[0].GetIpProperties()

IsDnsEnabled : False
IsDynamicDnsEnabled : True
DnsSuffix : novoclouds.com
AnycastAddresses : {}
UnicastAddresses : {System.Net.NetworkInformation.SystemUnicastIP}
MulticastAddresses : {System.Net.NetworkInformation.SystemMulticastIP}
DnsAddresses : {10.10.15.1, 10.200.1.21}
GatewayAddresses : {System.Net.NetworkInformation.SystemGatewayIP}
DhcpServerAddresses : {10.10.15.1}
WinsServersAddresses : {10.10.15.1}

PS C:\user> $adapters[0].GetPhysicalAddress()
B8AC6F8CD899
PS C:\user> ($adapters[0].GetIpProperties()).UnicastAddresses

Address : 10.10.90.59
IPv4Mask : 255.255.0.0
PrefixLength : 16
IsTransient : False
IsDnsEligible : True
PrefixOrigin : Dhcp
SuffixOrigin : OriginDhcp
DuplicateAddressDetectionState : Preferred
AddressValidLifetime : 1181907
AddressPreferredLifetime : 1181907
DhcpLeaseLifetime : 3456001
```

Operators

Friday, February 13, 2015
8:55 AM

Operator allow operations against variables including arithmetic, string, and other actions. PowerShell implements *overloading* of operators. This means the one operator may function differently based upon the data types of the operands.

Operator	Description	Example	Result
+	Addition	5 + 4.5	9.5
		2gb + 120mb	2273312768
		0x100 + 5	261
	Concatenation	"Hello " + "there"	"Hello there"
-	Subtraction	5 - 4.5	0.5
		12gb - 4.5gb	8053063680
		200 - 0xAB	29
*	Multiplication	5 * 4.5	22.5
		4mb * 3	12582912
		12 * 0xC0	2304
	Repetition	"x" * 5	"xxxxx"
/	Division	5 / 4.5	1.11111111111111
		1 mb / 30kb	34.13333333333333
		0xFFAB / 0xC	5454.25
	Modulus or Mod (remainder from integer division)		
		11 % 5	1

++	Increment operator	\$a++	Increment the value contained in \$a by 1
+=	Increment operator	\$a += 5	Add 5 to the current value contained in \$a
--	Decrement operator	\$a--	The value contained in \$a is decremented by 1
-=	Decrement operator	\$a -= 3	Subtract 3 from the current value contained in \$a
..	Range operator	10..19	Generates an array of ten elements containing the integers 10 through 19

We can use get-member to discover the type.	What happens when we mix types?
<pre>\$qty = 10 \$qty gm \$price = 1.50 \$price gm</pre>	<pre>\$ext = \$qty * \$price \$ext gm \$ext.gettype()</pre>
What happens when we explicitly type?	What happens if we don't know what we're doing?
[int] \$ext = \$qty * \$price \$ext	[int] \$i = 5GB
PS automatically recasts variables:	PS also has date-time types:
<pre>\$test = 10 \$test.gettype() \$test = 'it is now a string'</pre>	<pre>[datetime] \$d1 = get-date \$d1.gettype() [datetime] \$d2 = 'Jan 23 1949'</pre>

```
$test.GetType()
```

```
$d1 - $d2
```

The implicit typing of PowerShell produces some very interesting results when operators are used.

```
PS C:\Users>
PS C:\Users> ('3').GetType()
IsPublic IsSerial Name                                     BaseType
-----  -----  --  -----
True      True    String                                System.Object

PS C:\Users> '3' + 1
31
PS C:\Users> '3' * 5
33333
PS C:\Users> 1 + '3'
4
PS C:\Users> 5 * '3'
15
PS C:\Users> '3' / 3
1
PS C:\Users> -
```

As the illustration above shows, the order of the variables or literal values used in expressions and statements affects the type of the resultant object. Inspecting the above operations step-by-step:

1. The type of the *literal* string '3' (a literal is an expressed value) is a string. Regardless that this is a single character PowerShell considers this a string as shown by the GetType method. The syntax in the first statement creates the subexpression ('3') from which we reference the GetType method.
2. Since the + operator is overloaded (operates differently for different types), the expression '3' + 1 causes PowerShell to interpret the + operator as concatenation so PowerShell recasts the integer 1 to a string, concatenates it to '3' resulting in the string 31. This occurs because the string literal '3' begins the expression which PowerShell uses as a hint as to how the operator should be interpreted. Hence the result is a string and not an integer.
3. Similarly for the expression '3' * 5, PowerShell interprets the * operator as repetition and not multiplication and produces the string 33333.

4. In the expression `1 + '3'`, the integer `1` begins the expression causing PowerShell to interpret the `+` operator as addition. PowerShell then recasts the string literal `'3'` to an integer and performs the addition resulting in the integer `4`.
5. Since the division operator is not overloaded, the order of the variables doesn't matter. PowerShell simply recasts the string literal `'3'` to an integer and performs the division.

Many times when writing a script we need to implement loops to iterate through a collection of objects using one of the iteration (loop) operators (`foreach-object`, `foreach`, `for`, and `while`). A common function within the loop block is to count the number of objects processed and/or accumulate a total of one or more numeric properties.

The most common syntax for counting the number of objects processed in a loop is shown below.

```
$count = $count + 1      # count the items in the loop
```

This statement is interpreted as:

1. Retrieve the current value in `$count`
2. Add `1` to that value
3. Store the result back into `$count`

This operation is so common that the postfix increment operator `++` is implemented to reduce the amount of typing.

```
$count++      # count the items in the loop
```

Likewise, accumulating a total of a property or properties is accomplished by the syntax below. In this case, an object that has a `length` property is being processed in a pipeline.

```
$total = $total + $file.length # add the value of length property to  
$total
```

This statement is interpreted as:

1. Retrieve the current value in \$total
2. Retrieve the value of the property length in the object contained in \$file
3. Add both values together
4. Store the result back into \$total

This operation is also so common that the `+=` (plus assignment) operator is implemented to reduce the amount of typing.

```
$total += $file.length # add the value of length property to $total
```

While the measure-object cmdlet is available for counting and totaling, it may not be convenient or efficient to use for counting or accumulating values in many circumstances.

Arrays

Sunday, January 20, 2013
3:00 PM

Media: <https://www.youtube.com/watch?v=Ezh5OGLWT98>

An *array*, also called a *collection*, is a variable that has multiple values that may be referenced using an index. Arrays are *zero indexed* which means the first element is numbered 0 and not 1.

As seen previously, assigning a variable a comma-delimited list of values casts that variable to an array.

```
$a = 1,2,3,4,5,6,7,8,9,10  
$a  
$a.GetType()
```

```
PS C:\Users>  
PS C:\Users> $a = 1,2,3,4,5,6,7,8,9,10  
PS C:\Users> $a  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
PS C:\Users> $a.GetType()  
  
IsPublic IsSerial Name  
----- ----- --  
True     True    Object[]  
  
PS C:\Users>
```

BaseType

System.Array

The @ operator is another way of creating an array. In the assignment statement below \$a is initialized as an integer array of values 1 through 10. This is an identical result as that produced in the previous example.

```
$a = @(1,2,3,4,5,6,7,8,9,10)  
$a  
$a.GetType()
```

```
PS C:\Users>
PS C:\Users> $a = @(1,2,3,4,5,6,7,8,9,10)
PS C:\Users> $a
1
2
3
4
5
6
7
8
9
10
PS C:\Users> $a.GetType()
IsPublic IsSerial Name
----- -----
True     True      Object[]
```

BaseType

System.Array

One may wonder as to why this syntax should be used since the more terse syntax used in the previous example is available. The @ operator is handy when a null array needs to be initialized as below.

```
PS C:\Users>
PS C:\Users> $a = @()
PS C:\Users> $a
PS C:\Users>
PS C:\Users> $a.GetType()
IsPublic IsSerial Name
----- -----
True     True      Object[]
```

BaseType

System.Array

The Name property identifies object type. Each element of an array has a type (integer, floating point, string, etc). An array has no specific type specific type every element of an array is not required to have the same type. An element of an array may also be an array. For these reasons, the type of an array is the non-specific Object type with the [] indicating that the array supports nested arrays.

The GetType method is convenient when used interactively, it may not be appropriate inside a script. PowerShell provides a more script friendly conditional testing syntax so that the variable may be tested dynamically, i.e., while the script is executing.

```
$a = 1,2,3,4,5,6,7,8,9,10
$a -is [array]

$b = 'foobar'
$b -is [array]
```

```
PS C:\Users>
PS C:\Users> $a = 1,2,3,4,5,6,7,8,9,10
PS C:\Users> $a -is [array]
True
PS C:\Users>
PS C:\Users> $b = 'foobar'
PS C:\Users> $b -is [array]
False
PS C:\Users>
```

The size of the arrays is provided by the length property. Arrays are zero indexed so the first element of the array is \$a[0] as shown below. Similarly the tenth element, index 9, is the last element of the array. Also note in the illustration below, that you may index outside the bounds of the array as shown by \$a[10]. In which case PowerShell does not *throw an error* but returns a null value.

```
$a.length #size of the array
$a[0]      # first element of the array
$a[9]      # last element of the array
```

```
PS C:\Users>
PS C:\Users> $a.Length
10
PS C:\Users> $a[0]
1
PS C:\Users> $a[9]
10
PS C:\Users> $a[10]
PS C:\Users>
```

Arrays may also be negatively indexed as shown below, in this case, PowerShell just wraps around the array.

```
PS C:\Users>
PS C:\Users> $a[-1]
10
PS C:\Users> $a[-10]
1
PS C:\Users>
```

The example below shows how individual elements are referenced and how array arithmetic is done. Adding elements of an array requires referencing elements individually.

```
$sum = $a[0] + $a[1] + $a[2]
```

```
PS C:\Users> $a
1
2
3
4
5
6
7
8
9
10
PS C:\Users> $a[0]
1
PS C:\Users> $a[1]
2
PS C:\Users> $a[2]
3
PS C:\Users> $sum = $a[0] + $a[1] + $a[2]
PS C:\Users>
PS C:\Users> write-host $sum
6
PS C:\Users>
```

When working with large arrays, referencing elements individually may not be possible and certainly is not efficient from a script perspective. In a later section, we will learn how to use iteration (loops) to process the objects contained in an array.

Once an array is created, we may need to append elements to the array. Appending elements requires using the overloaded + operator.

```
$a
$a.length

$a = $a + 20

$a
$a.length
```

```
PS C:\Users>
PS C:\Users> $a
1
2
3
4
5
6
7
8
9
10
PS C:\Users> $a.length
10
PS C:\Users>
PS C:\Users> $a = $a + 20
PS C:\Users>
PS C:\Users> $a
1
2
3
4
5
6
7
8
9
10
20
PS C:\Users> $a.length
11
PS C:\Users>
```

We may also join multiple arrays together using the + or += operators as in the example below.

```
$a.length
$b = 30, 31, 32
$c = 40, 41, 42

$a = $a + $b
$a
```

```
$a += $c  
$a  
$a.length
```

```
PS C:\Users>  
PS C:\Users> $a.length  
11  
PS C:\Users> $b = 30, 31, 32  
PS C:\Users> $c = 40, 41, 42  
PS C:\Users>  
PS C:\Users> $a = $a + $b  
PS C:\Users> $a  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
20  
30  
31  
32  
PS C:\Users>  
PS C:\Users> $a += $c  
PS C:\Users> $a  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
20  
30  
31  
32  
40  
41  
42  
PS C:\Users> $a.length  
17  
PS C:\Users>
```

We may also declare an *explicitly typed* array as in the example below which creates an array of integers. In this case, all elements of the array must be of the same type. Notice that GetType shows that array type is Int32. Any value assigned to an element of the array must be an integer or must be in a form that can be cast as an integer. Note how in the execution illustration the floating point value 15.7 and the string '3.5' are accepted and appended to the array after rounding up. The string 'This causes an error' throws an error because it cannot be converted to an integer.

```
[int[]] $a = 1    # create an integer array with one element
$a
$a.length
$a.GetType()

$a += 15.7
$a

$a += '3.5'
$a

$a += 'This causes and error'

$a
```

```

PS C:\Users>
PS C:\Users> [int[]] $a = 1    # create an integer array with one element
PS C:\Users> $a
1
PS C:\Users> $a.Length
1
PS C:\Users> $a.GetType()


| IsPublic | IsSerial | Name    | BaseType     |
|----------|----------|---------|--------------|
| True     | True     | Int32[] | System.Array |



PS C:\Users>
PS C:\Users> $a += 15.7
PS C:\Users> $a
1
16
PS C:\Users>
PS C:\Users> $a += '3.5'
PS C:\Users> $a
1
16
4
PS C:\Users>
PS C:\Users> $a += 'This causes and error'
Cannot convert value "This causes and error" to type "System.Int32". Error: "not in a correct format."
At line:1 char:1
+ $a += 'This causes and error'
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException

PS C:\Users>
PS C:\Users> $a
1
16
4
PS C:\Users>

```

In the example below, we first create an array with no elements, i.e., a *null array*. Notice that GetType identifies the variable as an array. When \$b is piped into get-member, the cmdlet throws an error because no elements exist so the type is unknown which means the properties and methods are also unknown.

```
$b = @()
$b.GetType()
```

```
$b | gm
```

```
PS C:\user>
PS C:\user> $b = @()
PS C:\user> $b.GetType()

IsPublic IsSerial Name                                     BaseType
-----  -----  ----
True      True    Object[]                               System.Array

PS C:\user> $b | gm
gm : You must specify an object for the Get-Member cmdlet.
At line:1 char:6
+ $b | gm
+           ~~~
+ CategoryInfo          : CloseError: () [Get-Member], InvalidOperationException
+ FullyQualifiedErrorId : NoObjectInGetMember,Microsoft.PowerShell.Commands.GetMemberCommand
```

In the next step, we append the values 1 through 5 to the array. At the completion of this step, the array has five elements as indicated by displaying \$b.length. The type is still an array. Since we appended integer values, PowerShell was able to type each element of the array as an int32 as shown when \$b is piped into get-member.

```
$b += 1
$b += 2
$b += 3
$b += 4
$b.length
$b.GetType()
```

```

PS C:\user> $b = @()
PS C:\user> $b += 1
PS C:\user> $b += 2
PS C:\user> $b += 3
PS C:\user> $b += 4
PS C:\user> $b += 5
PS C:\user> $b.length
5
PS C:\user> $b.GetType()

IsPublic IsSerial Name                                     BaseType
----- -----   ----
True      True     Object[]                                System.Array

PS C:\user> $b | gm

TypeName: System.Int32

Name        MemberType Definition
----        ----- -----
CompareTo  Method    int CompareTo(System.Object value), int CompareTo(int
Equals     Method    bool Equals(System.Object obj), bool Equals(int obj),
GetHashCode Method   int GetHashCode()
GetType    Method   type GetType()
GetTypeCode Method   System.TypeCode GetTypeCode(), System.TypeCode IConve
.ToBoolean Method   bool IConvertible.ToBoolean(System.IFormatProvider pr
ToByte     Method   byte IConvertible.ToByte(System.IFormatProvider provi
ToChar     Method   char IConvertible.ToChar(System.IFormatProvider provi
.ToDateTime Method   datetime IConvertible.ToDateTime(System.IFormatProvid
.ToDecimal Method decimal IConvertible.ToDecimal(System.IFormatProvider
ToString   Method   string IConvertible.ToString(System.IFormatProvider)

```

Arrays, like scalar variables, are *polymorphic* (many forms). This means that elements in an array need not be all of the same type. In the example below, we create an array of zero elements, i.e., a null array. The string 'text' is appended becoming the first element of the array. The hexadecimal number 0xFF is appended as the second element of the array. The third element is the ASCII (Unicode) character 10 which is the new-line character. Note, new-line character is a non-display character which will appear in the output as a blank line. The fourth element is a string. And, finally the last element is the character c.

```

$a = @()
$a.length
$a = $a + 'text'
$a = $a + 0xFF

```

```
$a[0].GetType()  
$a[1].GetType()
```

```
PS C:\user> $a = @()
PS C:\user> $a.Length
0
PS C:\user> $a = $a + 'text'
PS C:\user> $a = $a + 0xFF
PS C:\user> $a[0].Gettype()

IsPublic IsSerial Name                                     BaseType
-----  -----  --
True      True    String                               System.Object

PS C:\user> $a[1].Gettype()

IsPublic IsSerial Name                                     BaseType
-----  -----  --
True      True    Int32                                System.ValueType
```

```
$a += [char] 10      # the new-line character
$a += 'after nl'
$a += [char] 'c'
$a.length
$a
```

Below are multiple assignment statements that create arrays showing the possible syntax that may be used.

```
$anArray = 1,"Hello",3.5,"World"  
$anotherArray = 5,10,17,19  
$oneMoreArray = @(1,"Hello",3.5,"World")
```

One final note on creating arrays. Arrays may be created implicitly when objects returned by cmdlets are assigned to a variable as shown below. In the first execution of get-process only one instance of Notepad is running so get-process returns one object making \$p a scalar variable as shown by Gettype. Since it is a scalar, PowerShell shows the type as "Process". In the second execution of get-process, multiple svchost processes are executing, so the cmdlet returns multiple objects causing PowerShell to type \$p as an array.

```
$p = get-process notepad  
$p  
$p.GetType()  
$p = get-process svchost  
$p  
$p.GetType()
```

```

PS C:\Users>
PS C:\Users> $p = get-process notepad
PS C:\Users> $p

```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
78	7	1356	5812	107	0.02	3588	notepad

```
PS C:\Users> $p.GetType()
```

IsPublic	IsSerial	Name	BaseType
True	False	Process	System.ComponentModel.Component

```

PS C:\Users> $p = get-process svchost
PS C:\Users> $p

```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
724	18	8480	12452	67		112	svchost
525	20	7360	14612	53		984	svchost
866	31	18660	26976	90		1060	svchost
779	37	22260	32264	111		1092	svchost
2172	66	32180	51432	209		1112	svchost
835	47	16612	27776	123		1200	svchost
706	69	12464	20640	1371		1460	svchost
639	40	25204	32812	116		1628	svchost
346	15	5608	13928	84		1876	svchost
102	8	1376	4928	22		2188	svchost
232	16	2572	8328	66		2248	svchost
552	29	7736	17080	69		3280	svchost
415	34	6188	13256	826		3852	svchost

```
PS C:\Users> $p.GetType()
```

IsPublic	IsSerial	Name	BaseType
True	True	Object[]	System.Array

An array may also be unbundled and each element distributed into individual scalar variables. In the example below, the elements of \$myArray are distributed to the scalar variables \$v1, \$v2, \$v3, and \$v4. The type of each variable is determined by the type of the corresponding array element.

```
$myArray = @(1,"Hello",3.5,"World")
$v1,$v2,$v3,$v4 = $myArray
```

```

PS C:\Users>
PS C:\Users> $myArray = @(1,"Hello",3.5,"World")
PS C:\Users> $myArray
1
Hello
3.5
World
PS C:\Users> $v1,$v2,$v3,$v4=$myArray
PS C:\Users> $v1
1
PS C:\Users> $v1.GetType()

IsPublic IsSerial Name                                     BaseType
-----  -----  --
True      True    Int32                                 System.ValueType

PS C:\Users> $v2
Hello
PS C:\Users> $v2.GetType()

IsPublic IsSerial Name                                     BaseType
-----  -----  --
True      True    String                                System.Object

PS C:\Users> $v3
3.5
PS C:\Users> $v3.GetType()

IsPublic IsSerial Name                                     BaseType
-----  -----  --
True      True    Double                               System.ValueType

PS C:\Users> $v4
World
PS C:\Users> $v4.GetType()

IsPublic IsSerial Name                                     BaseType
-----  -----  --
True      True    String                                System.Object

```

Working with Arrays:

The syntax in the following examples does the following:

1. Create an inline array using a subexpression and display the size of the array.

2. Create an inline array then append a new element in the array containing the value 7.
3. Create an inline array then append it to itself.
4. Create two inline arrays and append the second to the first.
5. Create an inline array of the specified characters.
6. Create an explicitly typed array and assign it a series of characters.

```
(1,2,3,4,5).length      # length of the array
(1,2,3,4,5) + 7        # create a new array containing the values 1 through 7
(1,2,3,4,5) * 2        # create a new array consisting of the array appended to itself
(1,2,3,4,5) + (6,7,8,9) # create a new array by joining both arrays
[char] 'a', [char] 'b', [char] 'c'    # create a new array of characters
[char[]] $chars =
'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u'
$chars
```

```
PS C:\Users>
PS C:\Users> (1,2,3,4,5).Length      # Length of the array
5
PS C:\Users> (1,2,3,4,5) + 7        # create a new array containing the values 1 through
1
2
3
4
5
7
PS C:\Users> (1,2,3,4,5) * 2        # create a new array consisting of the array appended
1
2
3
4
5
1
2
3
4
5
PS C:\Users> (1,2,3,4,5) + (6,7,8,9) # create a new array by joining both arrays
1
2
3
4
5
6
7
8
9
PS C:\Users> [char] 'a', [char] 'b', [char] 'c'    # create a new array of characters
a
b
c
PS C:\Users> [char[]] $chars = 'a','b','c','d','e','f','g','h','i','j','k','l','m','n','
PS C:\Users> $chars
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
```

Addressing an array:

```
[char[]] $chars =
'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u'
$chars[$chars.length - 1]      # Last element of the array
$chars[0]                      # first element of the array
$chars[9]                       # tenth element of the array
$chars[-1]                     # Last element of the array
$chars[-2]                     # Next to last element of the array
```

```
PS C:\Users>
PS C:\Users> [char[]] $chars = 'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u'
PS C:\Users> $chars[$chars.length - 1]      # Last element of the array
z
PS C:\Users> $chars[0]                      # first element of the array
a
PS C:\Users> $chars[9]                       # tenth element of the array
j
PS C:\Users> $chars[-1]                     # Last element of the array
z
PS C:\Users>
PS C:\Users> $chars[-2]                     # Next to last element of the array
y
```

Array *slices*:

An array slice is created by taking elements out of an existing array and creating a new array of those slices.

```
$chars[3..6]      # slice includes fourth through seventh value
$chars[0,2,4,6]   # slice contains first, third, fifth, and seventh
elements
```

```
PS C:\Users>
PS C:\Users> $chars[3..6]
d
e
f
g
PS C:\Users> $chars[0,2,4,6]
a
c
e
g
```

Removing elements from an array:

PowerShell has no explicit operator to remove elements of an array. However array slices may be used to accomplish the desired effect. The syntax below creates a new array consisting of the designated slices.

```
$newChars = $chars[0,2,4,6] + $chars[7..9]  
$newChars
```

```
PS C:\Users>  
PS C:\Users> $newChars = $chars[0,2,4,6] + $chars[7..9]  
PS C:\Users> $newChars  
a  
c  
e  
g  
h  
i  
j
```

Hash Tables

Sunday, January 20, 2013
8:35 PM

Hash tables allow mapping a set of keys to a set of values. Each key has a corresponding value. The set of key and value is called a *key-value pair*. Conceptually, hash tables are similar to arrays but instead of an integer index, the index into the hash table, called a *key*, maybe alphanumeric or numeric.

<http://technet.microsoft.com/en-us/library/ee692803.aspx>

In this example, the first statement creates a hash table with State name as the *key* and the State capital as the *value* and assign it to the variable \$states. We then display the contents of the variable which shows the keys (name0 and the corresponding values. The next two statements demonstrate how the value of the corresponding individual key may be displayed just by indexing into the has

table with the key. The item method may also be used to query the value for a specific key.

```
$states = @{"Washington" = "Olympia"; "Oregon" = "Salem";
"California" = "Sacramento"}
$states
$states['Oregon']
$states["California"]
$states.item('Washington')
```

```
PS C:\Users>
PS C:\Users> $states = @{"Washington" = "Olympia"; "Oregon" = "Salem"; "California" = "Sacramento"}
PS C:\Users> $states
Name          Value
----          -----
Washington    Olympia
Oregon        Salem
California   Sacramento

PS C:\Users> $states['Oregon']
Salem
PS C:\Users> $states["California"]
Sacramento
PS C:\Users> $states.item('Washington')
Olympia
```

Entries may be added to the hash table using either the add method or by simply indexing with the new key and assigning the new value.

```
$states.add("Michigan", "Lansing")
$states["Idaho"] = "Boise"
$states["Russia"] = "St Petersburg"
$states
```

```

PS C:\Users> $states.add("Michigan","Lansing")
PS C:\Users> $states["Idaho"] = "Boise"
PS C:\Users> $states["Russia"] = "St Petersburg"
PS C:\Users> $states

Name                           Value
----                          -----
Oregon                         Salem
Russia                         St Petersburg
Idaho                          Boise
Michigan                       Lansing
California                     Sacramento
Washington                     Olympia

```

Similarly, we may update existing has table entries by using the `set_item` method or directly indexing with the existing key and setting the value.

```

$states["Russia"]
$states.set_item("Russia","Moscow")
$states["Russia"]
$states["Russia"] = "St. Petersburg"
$states

```

```

PS C:\Users>
PS C:\Users> $states["Russia"]
St Petersburg
PS C:\Users> $states.set_item("Russia","Moscow")
PS C:\Users> $states["Russia"]
Moscow
PS C:\Users> $states["Russia"] = "St Petersburg"
PS C:\Users> $states["Russia"]
St Petersburg

```

Correspondingly entries may be removed using the `remove` method.

```

$states
$states.remove("Russia")
$states

```

```

PS C:\Users>
PS C:\Users> $states
Name                Value
----              -----
Washington          Olympia
Oregon              Salem
California          Sacramento
Russia              St Petersburg

PS C:\Users> $states.remove("Russia")
PS C:\Users> $states
Name                Value
----              -----
Washington          Olympia
Oregon              Salem
California          Sacramento

```

A few other methods are available when working the hast tables. The Clear method removes every entry in the hash table. The ContainsKey method tests for the existence of a key. Similarly the ContainsValue method tests for the existence of a specific value. The Count method provides the number of entries in the hast table.

```

$states
$states.Clear()
$states
$states = @{"Washington" = "Olympia"; "Oregon" = "Salem";
"California" = "Sacramento"; "Michigan" = "Lansing"}
$states.ContainsKey('Michigan')
$states.ContainsValue('Salem')
$states.Count

```

```

PS C:\Users>
PS C:\Users> $states
Name                Value
----              -----
Washington          Olympia
Michigan            Lansing
Oregon              Salem
California          Sacramento

PS C:\Users> $states.Clear()
PS C:\Users> $states
PS C:\Users> $states = @{ "Washington" = "Olympia"; "Oregon" = "Salem"; "California" = "Sacramento"; "Michigan" = "Lansing" }
PS C:\Users> $states.ContainsKey('Michigan')
True
PS C:\Users> $states.ContainsValue('Salem')
True
PS C:\Users> $states.Count
4

```

Using hash tables as parameters:

Many times hash tables may be used as parameters for a cmdlet as in the example below. In that example, a new derived property is created called Capacity. This property is calculated as the sum of the free (space) property and the used (space) property.

```

get-psdrive c |
    select-object name, free, used, @{Name='Capacity';Expression={ $_.free + $_.used }}

```

```

PS C:\Users>
PS C:\Users> get-psdrive c |
>>>     select-object name, free, used, @{Name='Capacity';Expression={ $_.free +
Name      Free       Used      Capacity
----      ---       ---      -----
C        225405919232 273872490496 499278409728

```

```

get-psdrive c | select-object name, free, used |
    format-table name, free, used, @{Label='Capacity';Expression={ $_.free + $_.used }}

```

```
PS C:\Users>
PS C:\Users> get-psdrive c | select-object name, free, used |
>>> format-table name, free, used, @{Label='Capacity';Expression={ $_.free + .used * 1GB }} | Out-GridView
Name      Free      Used      Capacity
----      ----      ----      -----
C    225405874176 273872535552 499278409728
```

Automatic Variables

Thursday, January 17, 2013
11:40 AM

Automatic variables are maintained by PowerShell to provide state information about PowerShell. More information is available by using the help system as below:

```
help about_automatic_variables
```

Some of the more interesting automatic variables are \$error, \$lastexitcode, \$PSVersionTable

Error history:

\$error is an array that contains the most recent errors. In the illustration below, the array \$a does not exist so when the assignment of a value to the first element in the array (\$a[0]) is attempted, PowerShell throws an error.

```
PS C:\Users>
PS C:\Users> $a[0] = 1
Cannot index into a null array.
At line:1 char:1
+ $a[0] = 1
+ ~~~~~
+ CategoryInfo          : InvalidOperationException: () [], RuntimeException
+ FullyQualifiedErrorId : NullArray
```

Likewise if \$s doesn't exist, or is not a string, attempting to call the Trim() method, which trims spaces from strings, throws an error.

```
PS C:\Users>
PS C:\Users> $s.Trim()
You cannot call a method on a null-valued expression.
At line:1 char:1
+ $s.Trim()
+ ~~~~~
+ CategoryInfo          : InvalidOperationException: () [], RuntimeException
+ FullyQualifiedErrorId : InvokeMethodOnNull
```

The above errors are stored in \$error array in order of most recent.

```
PS C:\Users>
PS C:\Users> $error[0]
You cannot call a method on a null-valued expression.
At line:1 char:1
+ $s.Trim()
+ ~~~~~
+ CategoryInfo          : InvalidOperationException: () [], RuntimeException
+ FullyQualifiedErrorId : InvokeMethodOnNull

PS C:\Users> $error[1]
Cannot index into a null array.
At line:1 char:1
+ $a[0] = 1
+ ~~~~~
+ CategoryInfo          : InvalidOperationException: () [], RuntimeException
+ FullyQualifiedErrorId : NullArray
```

Exit Code:

The notion of an exit code started early in the history of Unix scripting. Exit codes are a method for a program or script to provide information to the calling environment, i.e., the calling program or script, about the disposition of its exit,

i.e., normal or error. Windows uses this same notion with the *DOS external programs*, like ping, netstat, etc. As shown in the figure below, the *DOS* (cmd.exe) shell uses the pseudo environment variable %errorlevel to propagate the exit code. In examples below, a successful ping results in an exit code of 0 and an unsuccessful ping results in an exit code of 1. Typically, an exit code or errorlevel of 0 means normal exit anything else means the called program experienced some error.

```
C:\Users>ping 8.8.8.8 -n 1
Pinging 8.8.8.8 with 32 bytes of data:
Reply from 8.8.8.8: bytes=32 time=20ms TTL=55

Ping statistics for 8.8.8.8:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 20ms, Maximum = 20ms, Average = 20ms

C:\Users>echo %errorlevel%
0

C:\Users>ping 9.9.9.9 -n 1
Pinging 9.9.9.9 with 32 bytes of data:
Request timed out.

Ping statistics for 9.9.9.9:
    Packets: Sent = 1, Received = 0, Lost = 1 (100% loss),
C:\Users>echo %errorlevel%
1
```

PowerShell provides access to the exit code through the automatic variable \$lastexitcode which contains the exit code of the last *DOS* program executed from the PowerShell console. Very important: \$lastexitcode only applies to *DOS* external programs like ping or netstat. Repeating the same sequence of pings as done in the CMD window above shows that the value of the \$lastexitcode is consistent with the that of %errorlevel.

```
PS C:\Users>
PS C:\Users> ping 8.8.8.8 -n 1
Pinging 8.8.8.8 with 32 bytes of data:
Reply from 8.8.8.8: bytes=32 time=19ms TTL=55

Ping statistics for 8.8.8.8:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 19ms, Maximum = 19ms, Average = 19ms
PS C:\Users>
PS C:\Users> write-host $lastexitcode
0
PS C:\Users>
PS C:\Users> ping 9.9.9.9 -n 1
Pinging 9.9.9.9 with 32 bytes of data:
Request timed out.

Ping statistics for 9.9.9.9:
    Packets: Sent = 1, Received = 0, Lost = 1 (100% loss),
PS C:\Users>
PS C:\Users> write-host $lastexitcode
1
PS C:\Users>
```

Error Actions:

The automatic variable `$ErrorActionPreference` controls the behavior of PowerShell when errors are encountered. There are four values that may be assigned to this variable: `SilentlyContinue`, `Continue`, `Stop`, and `Inquire`. The default value is `Continue`.

As shown in the ISE image below, with `$ErrorActionPreference` set to the default value of `Continue`, the script throws the error but PowerShell continues executing the next cmdlet. Notice that the last `write-host` 'ending' cmdlet executed despite the occurrence of the error. Note, division by 0 is undefined in mathematics, hence anything divided by 0 in a script always throws an error.

The screenshot shows the Windows PowerShell ISE interface. The top menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. A toolbar with various icons is located above the main workspace. The left pane displays a script named Untitled1.ps1* with the following content:

```
1 # the default is $ErrorActionPreference = 'Continue'
2 write-host 'starting'
3 $i = 1/0
4 write-host 'ending'
```

The right pane shows the output of running the script in the PowerShell environment:

```
PS C:\> # the default is $ErrorActionPreference = 'Continue'
write-host 'starting'
$i = 1/0
write-host 'ending'
starting
Attempted to divide by zero.
At line:3 char:1
+ $i = 1/0
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], RuntimeException
+ FullyQualifiedErrorId : RuntimeException

ending
PS C:\>
```

The status bar at the bottom indicates "Completed" and shows the cursor position as "Ln 15 Col 9".

Setting \$ErrorActionPreference to stop causes the script to stop executing at the point of the error. Notice that in the image below the string 'ending' was not displayed since the script stopped as a result of the error.

The screenshot shows the Windows PowerShell ISE interface. The top menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. A code editor window titled "Untitled1.ps1*" contains the following PowerShell script:

```
1 $ErrorActionPreference = 'Stop'
2 write-host 'starting'
3 $i = 1/0
4 write-host 'ending'
```

Below the code editor is a command-line window showing the execution of the script. The output is:

```
PS C:\> $ErrorActionPreference = 'Stop'
write-host 'starting'
$i = 1/0
write-host 'ending'
starting
Attempted to divide by zero.
At Line:3 char:1
+ $i = 1/0
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : RuntimeException
```

After the error, the command line prompt PS C:\> | is visible.

The value `SilentlyContinue`, as the name implies, causes PowerShell to ignore the error and continue executing the script.

The screenshot shows the Windows PowerShell ISE interface. The top menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. A code editor window titled "Untitled1.ps1*" contains the following PowerShell script:

```
1 $ErrorActionPreference = 'SilentlyContinue'
2 write-host 'starting'
3 $i = 1/0
4 write-host 'ending'
```

Below the code editor is a large dark blue terminal window showing the execution of the script. The output is:

```
PS C:\> $ErrorActionPreference = 'SilentlyContinue'
write-host 'starting'
$i = 1/0
write-host 'ending'
starting
ending
```

At the bottom of the terminal window, there is a status bar with "Ln 8 Col 9" and a zoom level of "100%".

The value Inquire for \$ErrorActionPreference causes PowerShell to show a message with options for the user as to the disposition of the script.

The screenshot shows the Windows PowerShell Integrated Scripting Environment (ISE) interface. At the top, the menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. The main area contains a code editor window titled "Untitled1.ps1*" with the following PowerShell script:

```
1 $ErrorActionPreference = 'Inquire'
2 write-host 'starting'
3 $i = 1/0
4 write-host 'ending'
```

Below the code editor is a command-line window showing the execution of the script:

```
PS C:\> $ErrorActionPreference = 'Inquire'
write-host 'starting'
$i = 1/0
write-host 'ending'
starting
```

A modal dialog box titled "Action to take for this exception:" is displayed, containing the message "Attempted to divide by zero." with four buttons: Continue, Silently Continue, Break, and Suspend.

At the bottom of the interface, status bars indicate "Running script / selection. Press Ctrl+Break to stop." and "Ln 1 Col 34". The zoom level is set to 100%.

Since the action to be taken in the event of an error may be cmdlet-specific, each cmdlet implements the `-ErrorAction` parameter which may be set to any of the four values of `$ErrorPreference` and overrides the current setting of `$ErrorPreference` for the singular execution of the cmdlet. The example below demonstrates the functionality. In the first execution of `get-childitem`, an error is thrown since the user executing the cmdlet does not have permission to access the sub-directory `Telemetry`.

```

PS C:\Users>
PS C:\Users> get-childitem -path C:\windows\AppCompat -recurse | out-host -Paging

    Directory: C:\windows\AppCompat

Mode                LastWriteTime         Length Name
----                -----          ----  -
d----        7/13/2015  1:21 AM           0 Appraiser
d---s       7/15/2015  3:52 AM           0 Programs
d---       6/11/2015 12:22 AM           0 UA

    Directory: C:\windows\AppCompat\Appraiser

Mode                LastWriteTime         Length Name
----                -----          ----  -
d----        7/15/2015  1:30 AM           0 Gated
d---s       7/13/2015  1:20 AM           0 Telemetry

    Directory: C:\windows\AppCompat\Appraiser\Gated

Mode                LastWriteTime         Length Name
----                -----          ----  -
-a---        7/15/2015  1:30 AM        156 DailyGatedCheck.cab
-a---       6/29/2015  6:18 PM         96 gated.xml
get-childitem : Access to the path 'C:\windows\AppCompat\Appraiser\Telemetry' is denied.
At line:1 char:1
+ get-childitem -path C:\windows\AppCompat -recurse | out-host -Paging
+ ~~~~~~ + CategoryInfo          : PermissionDenied: (C:\windows\AppCompat\Appraiser\Telemetry:UnauthorizedAccessException) + FullyQualifiedErrorId : DirUnauthorizedAccessError,Microsoft.PowerShell.Commands.GetChildItemCommand
get-childitem : Access to the path 'C:\windows\AppCompat\Programs' is denied.
At line:1 char:1
+ get-childitem -path C:\windows\AppCompat -recurse | out-host -Paging
+ ~~~~~~ + CategoryInfo          : PermissionDenied: (C:\windows\AppCompat\Programs:String) + UnauthorizedAccessException + FullyQualifiedErrorId : DirUnauthorizedAccessError,Microsoft.PowerShell.Commands.GetChildItemCommand

    Directory: C:\windows\AppCompat\UA

Mode                LastWriteTime         Length Name
----                -----          ----  -
-a---        7/13/2015  1:11 AM      1192 GenericApp.png

```

Re-executing the get-childitem with -ErrorAction SilentlyContinue results in the elimination of the error display. Note, an error still occurred and was detected by the cmdlet; however, the cmdlet does not show the error on the console.

```

PS C:\Users>
PS C:\Users> get-childitem -path C:\windows\AppCompat -recurse -ErrorAction SilentlyContinue

    Directory: C:\windows\AppCompat

Mode                LastWriteTime      Length Name
----              -----          ----  --
d----
```

Mode	LastWriteTime	Length	Name
d----	7/13/2015 1:21 AM		Appraiser
d---s	7/15/2015 3:52 AM		Programs
d----	6/11/2015 12:22 AM		UA

```

    Directory: C:\windows\AppCompat\Appraiser

Mode                LastWriteTime      Length Name
----              -----          ----  --
d----
```

Mode	LastWriteTime	Length	Name
d----	7/15/2015 1:30 AM		Gated
d---s	7/13/2015 1:20 AM		Telemetry

```

    Directory: C:\windows\AppCompat\Appraiser\Gated

Mode                LastWriteTime      Length Name
----              -----          ----  --
-a---
```

Mode	LastWriteTime	Length	Name
-a---	7/15/2015 1:30 AM	156	DailyGatedCheck.cab
-a---	6/29/2015 6:18 PM	96	gated.xml

```

    Directory: C:\windows\AppCompat\UA

Mode                LastWriteTime      Length Name
----              -----          ----  --
-a---
```

Mode	LastWriteTime	Length	Name
-a---	7/13/2015 1:11 AM	1192	GenericApp.png

Confirmation:

Because the effect of some operations is critical and could result in disruption of system operations or loss of data, PowerShell implements \$ConfirmPreference.

Many cmdlets have a default setting for \$ConfirmPreference based upon their impact.

Under normal circumstances the following script would execute without issue:

```
remove-item *.txt
```

Changing \$ConfirmPreference causes the actions to be confirmed.

```
$ConfirmPreference = "Low"  
remove-item *.txt
```

```
PS C:\Users>  
PS C:\Users> $ConfirmPreference = "Low"  
PS C:\Users> remove-item *.txt  
  
Confirm  
Are you sure you want to perform this action?  
Performing the operation "Remove File" on target "C:\Users\in.txt".  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is
```

Other Automatic Variables:

The variable \$PSVersionTable is a read-only automatic variable that provides version information about PowerShell

```
PS C:\Users>  
PS C:\Users> $PSVersionTable  


| Name                      | Value                |
|---------------------------|----------------------|
| PSVersion                 | 4.0                  |
| WSManStackVersion         | 3.0                  |
| SerializationVersion      | 1.1.0.1              |
| CLRVersion                | 4.0.30319.34209      |
| BuildVersion              | 6.3.9600.17400       |
| PSCompatibleVersions      | {1.0, 2.0, 3.0, 4.0} |
| PSRemotingProtocolVersion | 2.2                  |


```

The \$home and the \$pwd variables identify the users home directory and the current directory respectively.

```
PS C:\Users>
PS C:\Users> $home
C:\Users\mike
PS C:\Users>
PS C:\Users> $pwd

Path
-----
C:\Users
```

The \$? Is similar to \$lastexitcode but for PowerShell cmdlets.

```
PS C:\Users>
PS C:\Users> test-connection 8.8.8.8 -count 1

Source          Destination        IPV4Address      IPV6Address
----          -----        -----
EL-CID          8.8.8.8          8.8.8.8

PS C:\Users> $?
True
PS C:\Users>
PS C:\Users> test-connection 9.9.9.9 -count 1
test-connection : Testing connection to computer '9.9.9.9' failed: Error
At line:1 char:1
+ test-connection 9.9.9.9 -count 1
+
+ CategoryInfo          : ResourceUnavailable: (9.9.9.9:String) [Test
+ FullyQualifiedErrorId : TestConnectionException,Microsoft.PowerShell
PS C:\Users> $?
False
PS C:\Users>
```

Working with [datetime]

Friday, August 28, 2015
3:51 PM

PowerShell provides rich functionality for working with date times.

A date-time object representing the current date may be retrieved using the get-date cmdlet. Piping the date-time object into format-list exposes all the properties of that object.

```
get-date
```

```
get-date | format-list
```

```
PS C:\Users>
PS C:\Users>
PS C:\Users> get-date
Thursday, January 14, 2016 3:22:31 PM

PS C:\Users>
PS C:\Users> get-date | format-list

DisplayHint : DateTime
Date        : 1/14/2016 12:00:00 AM
Day         : 14
DayOfWeek   : Thursday
DayOfYear    : 14
Hour        : 15
Kind         : Local
Millisecond : 454
Minute       : 22
Month        : 1
Second       : 31
Ticks        : 635883817514547267
TimeOfDay    : 15:22:31.4547267
Year         : 2016
DateTime     : Thursday, January 14, 2016 3:22:31 PM
```

Using `get-member` to inspect the properties and methods of a date-time object reveals some very useful properties and methods.

```
get-date | get-member
```

```

PS C:\Users>
PS C:\Users> get-date | get-member

TypeName: System.DateTime

Name          MemberType      Definition
----          -----
Add           Method         datetime Add(timespan value)
AddDays       Method         datetime AddDays(double value)
AddHours      Method         datetime AddHours(double value)
AddMilliseconds Method        datetime AddMilliseconds(double value)
AddMinutes    Method         datetime AddMinutes(double value)
AddMonths    Method         datetime AddMonths(int months)
AddSeconds    Method         datetime AddSeconds(double value)
AddTicks      Method         datetime AddTicks(long value)
AddYears      Method         datetime AddYears(int value)
CompareTo     Method         int CompareTo(System.Object value), int CompareTo(datetime value)
Equals        Method         bool Equals(System.Object value), bool Equals(datetime value), b
GetDateTimeFormats Method        string[] GetDateTimeFormats(), string[] GetDateTimeFormats(Syste
GetHashCode   Method         int GetHashCode()
GetObjectData Method        void ISerializable.GetObjectData(System.Runtime.Serialization.Ser

```

We may use these methods to find dates in the future or the past. The examples below show how the AddHours and AddYears method may be used to derive days in the past the future.

```

$dt = get-date

$dt
# three years into the future
$dt.AddYears(3)
# five years into the past
$dt.AddYears(-5)
# 12 hours into the future
$dt.AddHours(12)
# 120 hours in the past
$dt.AddHours(-120)

```

```
PS C:\Users>
PS C:\Users> $dt = get-date
PS C:\Users>
PS C:\Users> $dt
Thursday, January 14, 2016 3:18:47 PM

PS C:\Users> # three years into the future
PS C:\Users> $dt.AddYears(3)
Monday, January 14, 2019 3:18:47 PM

PS C:\Users> # five years into the past
PS C:\Users> $dt.AddYears(-5)
Friday, January 14, 2011 3:18:47 PM

PS C:\Users> # 12 hours into the future
PS C:\Users> $dt.AddHours(12)
Friday, January 15, 2016 3:18:47 AM

PS C:\Users> # 120 hours in the past
PS C:\Users> $dt.AddHours(-120)
Saturday, January 09, 2016 3:18:47 PM
```

Properties are also available that deconstruct the date into its component parts.

```
# Day of the week
$dt.DayofWeek
# Day of the year
$dt.DayofYear
# Month
$dt.Month
# Day of month
$dt.Day
# Year
$dt.Year
```

```
PS C:\Users>
PS C:\Users> # Day of the week
PS C:\Users> $dt.DayofWeek
Thursday
PS C:\Users> # Day of the year
PS C:\Users> $dt.DayofWeek
Thursday
PS C:\Users> # Month
PS C:\Users> $dt.Month
1
PS C:\Users> # Day of month
PS C:\Users> $dt.Day
14
PS C:\Users> # Year
PS C:\Users> $dt.Year
2016
PS C:\Users>
```

The script developer may explicitly type a variable as [datetime]. Any string that is a from that could be interpreted as a date can be converted to a [datetime] type. The example below demonstrates two methods for coercing a string into a [datetime] type. While the result is the same, the operation is different. In the first example, PowerShell creates the [datetime] variable \$mydate then converts the string into a [datetime] type and upon successful conversion assigns the new [datetime] object derived from the string to \$mydate. In the second example, PowerShell converts the string to a new [datetime] object then creates \$mydate as a [datetime] time and assigns the convert object.

```
[datetime] $mydate = 'Jan 1, 2018'
$mydate.GetType()

[datetime] $mydate = '12/7/194'
$mydate.GetType()
```

```

PS> [datetime] $mydate = 'Jan 1, 2018'
PS> $mydate.GetType()
IsPublic IsSerial Name                                     BaseType
----- ----- ----
True      True    DateTime                               System.ValueType

PS> $mydate = [datetime] '12/7/1941'
PS> $mydate.GetType()
IsPublic IsSerial Name                                     BaseType
----- ----- ----
True      True    DateTime                               System.ValueType

```

What is most useful is the date-time arithmetic available to the script writer. In the example below, we cast to date strings into date-time objects. We first cast a string that represents into a data-time object and assign it to the variable \$d1. Implicitly, \$d1 is now a date-time object. In the next statement, the string containing the date is assign to \$d2 which is a data-time type. This cause PowerShell to attempt to cast the string into a date-time object which it does successfully.

Notice that the format of the date strings in the assignment statement differs. PowerShell is able to convert any string that looks like a date into a date-time object. Also, since the strings did not specify hours, minutes, or seconds; the time associated with the \$dt variable is 12:00:00 AM.

We then subtract \$d1 from \$d2. This results in yet another object called a *timespan* object. The properties of the time span object are shown in the display.

```

# Day of the week
$d1 = [datetime] 'January 14, 2016'
[datetime] $d2 = '2-17-2018'
$d2 - $d1

```

```
PS C:\Users>
PS C:\Users> $d1 = [datetime] 'January 14, 2016'
PS C:\Users>
PS C:\Users> [datetime] $d2 = '2-17-2018'
PS C:\Users>
PS C:\Users> $d2 - $d1

Days          : 765
Hours         : 0
Minutes       : 0
Seconds       : 0
Milliseconds  : 0
Ticks         : 6609600000000000
TotalDays     : 765
TotalHours    : 18360
TotalMinutes  : 1101600
TotalSeconds  : 66096000
TotalMilliseconds : 66096000000
```

This type of arithmetic is useful in many cases; one example is find files created in the past. This example use date-time arithmetic to find the number of days in the past the file was created.

The first statement creates a *FileInfo* object that represents the file servers.csv on the file system. That object has everything you would want to know about the file. We then display the creation date-time on the console. We then do date arithmetic to calculate the number of days in the past the file was created. Note, the syntax (get-date) is called a subexpression. Enclosing a PowerShell expression in parentheses as below causes PowerShell to evaluate the syntax with the subexpression and produce the resulting object. Omitting the parentheses would result in a syntax error.

```
# get the fileInfo object representing the file servers.csv
$f = get-childitem servers.csv

# show the date-time the file was created
$f.CreationTime

# How many days ago was the file created
(get-date) - $f.CreationTime
```

```
PS C:\Users>
PS C:\Users> # get the fileInfo object representing the file servers.csv
PS C:\Users> $f = get-childitem servers.csv
PS C:\Users>
PS C:\Users> # show the date-time the file was created
PS C:\Users> $f.CreationTime

Friday, August 21, 2015 11:23:33 AM

PS C:\Users>
PS C:\Users> # How many days ago was the file created
PS C:\Users> (get-date) - $f.CreationTime

Days : 146
Hours : 4
Minutes : 21
Seconds : 23
Milliseconds : 445
Ticks : 126300834454735
TotalDays : 146.181521359647
TotalHours : 3508.35651263153
TotalMinutes : 210501.390757892
TotalSeconds : 12630083.4454735
TotalMilliseconds : 12630083445.4735
```

Text Strings

Monday, April 15, 2013
7:52 AM

PowerShell is an interpretative language that, at times, can recognize that a value or series of values is a string. This is best illustrated by the example below. In the first execution, PowerShell recognized that the text following the write-host cmdlet is a text string. The execution is the conventional way of representing a text string where the string is delimited by quotes. As a general rule always delimit text strings by either a *single quote* or *double quote*.

```
PS>
PS> Write-Host the quick brown
the quick brown
PS>
PS> Write-Host 'the quick brown'
the quick brown
PS>
```

Single quotes delimit a text literal. Double quotes delimit a literal but causes PowerShell to inspect the text string for variables to expand.

Quoting:

Consider the following example where PowerShell considers everything delimited by single quotes as text and displays it accordingly. Observe that when the single quotes are substituted with double quotes, PowerShell inspects that contents of the text string to determine if expansion is necessary.

```
PS C:\Users> $price = .10
PS C:\Users> $qty = 1000
PS C:\Users> $ext = $price * $qty
PS C:\Users> write-host 'The price $price * the quantity $qty is $ext'
The price $price * the quantity $qty is $ext
PS C:\Users>
PS C:\Users> write-host "The price $price * the quantity $qty is $ext"
The price 0.1 * the quantity 1000 is 100
PS C:\Users> -
```

For the sake of completeness, we can eliminate the need to the \$ext variable by using the sub-expression `$($price * $qty)` as shown below. The `"$("` and `")"` delimit the subexpression which causes PowerShell to execute the script code between the delimiters.

```
PS C:\Users>
PS C:\Users> write-host "The price $price * the quantity $qty is $($price * $qty)
The price 0.1 * the quantity 1000 is 100
PS C:\Users>
```

Interpreting Subexpressions

If delimited by double quotes, the text literal is interpreted by scanning for any variable or sub-expressions prefixed by `$` or for the back tick (```) which can act as an escape character or as a prefix for special characters.

```
"The Windows directory is here: $env:windir"  
"The host name is $env:computername"
```

```
PS C:\Users>  
PS C:\Users> "The Windows directory is here: $env:windir"  
The Windows directory is here: C:\Windows  
PS C:\Users>  
PS C:\Users> "The host name is $env:computername"  
The host name is MINK
```

This works well with simple variables. But when using object properties are used in double-quoted strings the results is not what is expected as shown in the next example.

```
PS>  
PS> $file = get-childitem C:\windows\notepad.exe  
PS>  
PS> $file.length  
246784  
PS> $file.name  
notepad.exe  
PS>  
PS> write-host "This file $file.name has a length of $file.length"  
This file C:\windows\notepad.exe.name has a length of C:\windows\notepad.exe.Length  
PS>  
PS>
```

The color coding gives a hint as to how PowerShell is interpreting the object property. Note that in the string all text is colored blue and the variables are colored green. Note also that the property names name and length are colored blue meant that PowerShell is only expanding the variable \$file and is considering name and length as text with the string. To alter this behavior, we need to create a subexpression containing the object properties. While the normal delimiters for subexpressions are parentheses, a subexpression used with a double-quoted string requires that the opening delimiter be "\$(" and not just "(".

```
PS> $file = get-childitem C:\windows\notepad.exe
PS>
PS> $file.length
246784
PS> $file.name
notepad.exe
PS>
PS> write-host "This file $file.name has a length of $file.length"
This file C:\windows\notepad.exe.name has a length of C:\windows\notepad.exe.length
PS>
PS>
PS> write-host "This file $($file.name) has a length of $($file.length)"
This file notepad.exe has a length of 246784
PS>
```

Note that the property names name and length are now colored white indicating the PowerShell understands that they are property names of the object contained in the variable.

This example takes the notion of the quoted subexpression to the extreme. PowerShell executes the pipeline `get-childitem | measure-object` which retrieves all of the files and subdirectories in the current directory and pipes the associated `FileInfo` or `DirectoryInfo` object into `measure-object`. The cmdlet `measure-object` counts the number of objects it receives and creates `MeasureInfo` object with the count. PowerShell then retrieves the value of the `Count` property and places it in the string. That string is then passed to `write-host` to display on the console.

```
PS>
PS>
PS> write-host "There are $( (get-childitem | measure-object).Count ) files in this directory"
There are 20 files in this directory
PS>
```

Metacharacters

Other metacharacters that may be embedded in text strings are show below.

Metacharacter	Definition
`n	New line
`r	Carriage return
`t	Tab
`a	Alarm (bell)
`b	Backspace
`"	Single quote
`"	Double quote
`0	Null
``	Backtick

Metacharacters may be used for formatting output intended for the console.
Below are examples of usage.

Displaying output on multiple lines.

```
PS>
PS> write-host "Line 1 of the output`nLine 2 of the output`nLine 3 of the output"
Line 1 of the output
Line 2 of the output
Line 3 of the output
PS>
```

Display columnar output.

```
PS>
PS>
PS> write-host "Row1Col1`tRow1Col2`nRow2Col1`tRow2Col2`nRow3Col1`tRow3Col2"
Row1Col1      Row1Col2
Row2Col1      Row2Col2
Row3Col1      Row3Col2
PS>
```

Here string:

A here string is a single-quoted or double-quoted string which can span multiple lines. All the lines in a here-string are interpreted as strings, albeit they are not delimited by quotes.

```
$mycolor = 'brown'

$aHereString = @"
The quick
$mycolor
fox jumps over the lazy dog
"@"

write-host $aHereString
```

```
PS C:\Users>
PS C:\Users> $mycolor = 'brown'
PS C:\Users>
PS C:\Users> $aHereString = @"
>>> The quick
>>> $mycolor
>>> fox jumps over the lazy dog
>>> "@
PS C:\Users>
PS C:\Users> write-host $aHereString
The quick
brown
fox jumps over the lazy dog
```

String Operators

Tuesday, April 16, 2013

5:46 AM

Replacing text in a string:

There are two ways to replace text in a string: use the *replace* string method or the *-replace* PowerShell operator.

In this first example, we use the *replace* method to replace substrings within the original string. This example also shows the use of the metacharacters `t, `n, and ``.

```
$str = 'The quick brown fox jumps over the lazy dog'  
$newstr = $str.replace('brown','red')  
write-host "`tThe old string was`"$str`" `n`tThe new string is  
`"$newstr`"" -foreground magenta
```

```
|PS C:\Users>  
PS C:\Users> $str = 'The quick brown fox jumps over the lazy dog'  
PS C:\Users> $newstr = $str.replace('brown','red')  
PS C:\Users> write-host "`tThe old string was`"$str`" `n`tThe new string is `"$newstr`"  
|       The old string was"The quick brown fox jumps over the lazy dog"  
|       The new string is "The quick red fox jumps over the lazy dog"  
PS C:\Users>
```

The *-replace* operator replaces substrings within strings. The left side of the operator is the string to be modified. The right side of the operator consists of two substrings. The first substring is the match substring. The second substring is the replacement. If the match substring is found by the *replace* operator, it is replaced by the second substring.

```

# replace brown with green
$s = 'The quick brown fox jumps of the lazy dog' -replace 'brown',
'green'
write-host $s -foreground magenta

# replace 'quick brown fox' by ' big green frog'
$s = 'The quick brown fox jumps of the laze dog' -replace 'quick
brown fox', 'big green frog'
write-host $s -foreground magenta

# can use a sub-expression
$s = 'The quick brown fox jumps of the lazy dog' -replace 'brown',
"$(read-host 'what color')"
write-host $s -foreground magenta

```

```

PS C:\Users>
PS C:\Users> # replace brown with green
PS C:\Users> $s = 'The quick brown fox jumps of the lazy dog' -replace 'brown', 'gr
PS C:\Users> write-host $s -foreground magenta
The quick green fox jumps of the lazy dog
PS C:\Users>
PS C:\Users> # replace 'quick brown fox' by ' big green frog'
PS C:\Users> $s = 'The quick brown fox jumps of the laze dog' -replace 'quick brown
PS C:\Users> write-host $s -foreground magenta
The big green frog jumps of the laze dog
PS C:\Users>
PS C:\Users> # can use a sub-expression
PS C:\Users> $s = 'The quick brown fox jumps of the lazy dog' -replace 'brown', "$(
what color: yellow
PS C:\Users> write-host $s -foreground magenta
The quick yellow fox jumps of the lazy dog

```

The *-join* operator creates a string from an array.

```

$a = 'a', 'b', 'c', 'd', 'e'

# joins the array with no separators
$s = -join $a
write-host $s -foreground magenta

# joins the array with ':' separating each element of the array
$s = $a -join ':'
write-host $s -foreground magenta

```

```
PS C:\Users>
PS C:\Users> $a = 'a', 'b', 'c', 'd', 'e'
PS C:\Users>
PS C:\Users> # joins the array with no separators
PS C:\Users> $s = -join $a
PS C:\Users> write-host $s -foreground magenta
abcde
PS C:\Users>
PS C:\Users> # joins the array with ':' separating each element of the array
PS C:\Users> $s = $a -join ':'
PS C:\Users> write-host $s -foreground magenta
a:b:c:d:e
```

The * operator is a string replicator. In the example below

```
# generate a string of six occurrences of the specified string
'abc' * 6

# generate a substring of five '0'
$str = 'Hell' + 'o' *5 + '!'
$str
```

```
PS C:\Users>
PS C:\Users> # generate a string of six occurrences of the specified string
PS C:\Users> 'abc' * 6
abcabcabcabcabcabc
PS C:\Users>
PS C:\Users> # generate a substring of five '0'
PS C:\Users>
PS C:\Users> $str = 'Hell' + 'o' *5 + '!'
PS C:\Users> $str
Hellooooo!
```

The more interesting string operators are the methods of the [string] class.

Splitting a string:

The split operator splits a string into an array of substrings based upon a delimiter.

```
$strscalar = 'a,b,c,d,e,f'
$strarray = $strscalar.split(',')
write-host "The are $($strarray.length) elements" -ForegroundColor Magenta
write-host "Then contents of the string array are $($strarray)" -ForegroundColor Magenta
```

```
$strarray
```

```
PS C:\Users>
PS C:\Users> $strscalar = 'a,b,c,d,e,f'
PS C:\Users> $strarray = $strscalar.split(',')
PS C:\Users> write-host "The are $($strarray.length) elements" -ForegroundColor Magenta
The are 6 elements
PS C:\Users> write-host "Then contents of the string array are $($strarray)" -ForegroundColor Magenta
Then contents of the string array are a b c d e f
PS C:\Users> $strarray
a
b
c
d
e
f
```

```
# Extract the OUI in a MAC address
# the long then the short way
$mac = '2C-30-68-C4-86-4D'
$octets = $mac.split('-') # split mac into array of octets
$oui = $octets[0..2] -join ':' # join the first three octets
# delimited by :
write-host The oui is $oui -foregroundcolor magenta

$mac = 'B8-AC-6F-8C-D8-99'
write-host "The oui is $($($mac.split('-'))[0..2] -join ':')"
-foregroundcolor magenta
```

```
PS C:\Users>
PS C:\Users> $mac = '2C-30-68-C4-86-4D'
PS C:\Users> $octets = $mac.split('-') # split mac into array of octets
PS C:\Users> $oui = $octets[0..2] -join ':' # join the first three octets
PS C:\Users> write-host The oui is $oui -foregroundcolor magenta
The oui is 2C:30:68
PS C:\Users>
PS C:\Users> $mac = 'B8-AC-6F-8C-D8-99'
PS C:\Users> write-host "The oui is $($($mac.split('-'))[0..2] -join ':')"
The oui is B8:AC:6F
PS C:\Users>
```

Comparing a string:

```
$str1 = 'The quick brown fox'
$str2 = 'The quick red fox'

$str1.CompareTo($str1)
$str1.CompareTo($str2)
```

```
($str1 -replace 'brown', 'red').CompareTo($str2)

$str1.StartsWith('The')
$str1.EndsWith('fox')

$str1.Contains('fox')
```

```
PS C:\Users>
PS C:\Users> $str1 = 'The quick brown fox'
PS C:\Users> $str2 = 'The quick red fox'
PS C:\Users>
PS C:\Users> $str1.CompareTo($str1)
0
PS C:\Users> $str1.CompareTo($str2)
-1
PS C:\Users>
PS C:\Users> ($str1 -replace 'brown', 'red').CompareTo($str2)
0
PS C:\Users>
PS C:\Users> $str1.StartsWith('The')
True
PS C:\Users> $str1.EndsWith('fox')
True
PS C:\Users>
PS C:\Users> $str1.Contains('fox')
True
```

Changing case of a string:

```
$str1.ToLower()

$str1.ToUpper()
```

```
PS C:\Users>
PS C:\Users> $str1.ToLower()
the quick brown fox
PS C:\Users>
PS C:\Users> $str1.ToUpper()
THE QUICK BROWN FOX
```

Extracting substrings:

Think of strings as an array of characters which are zero-indexed.

```
$str3 = 'The quick brown fox jumps of the lazy dog'

$str3.Substring(4)  # extract starting at the fifth position and
continuing until the end
```

```
$str3.Substring(4,5) # extract five characters starting at the fifth position
```

```
PS C:\Users>
PS C:\Users> $str3 = 'The quick brown fox jumps of the lazy dog'
PS C:\Users>
PS C:\Users> $str3.Substring(4)    # extract starting at the fifth position and consider the length
quick brown fox jumps of the lazy dog
PS C:\Users>
PS C:\Users> $str3.Substring(4,5) # extract five characters starting at the fifth position
quick
```

Converting strings to numbers:

PowerShell automatically converts literal strings to numbers whenever the string is assigned to a numeric type variable or when the string is cast as a number:

```
[int] $n = '1235'
$n
$m = [int] '1234'
$m
```

```
PS C:\Users>
PS C:\Users> [int] $n = '1235'
PS C:\Users> $n
1235
PS C:\Users> $m = [int] '1234'
PS C:\Users>
PS C:\Users> $m
1234
```

However there are instances when the contents of the string is unknown but assumed to be a number. To avoid throwing errors, the TryParse method of the [int] type may be used. TryParse will convert a string to a number if the string contents can be converted.

```
[int] $n = $null # define a variable to hold the result
[string] $s = read-host 'Enter a string'
$ok = [int]::TryParse($s, [ref] $n)
if ($ok) { write-host "The number is $n" -ForegroundColor cyan}
else { write-host 'conversion was not successful' -ForegroundColor red}
```

```

1 [int] $n = $null # define a variable to hold the result
2 [string] $s = read-host 'Enter a string'
3 $ok = [int]::TryParse($s, [ref] $n)
4 if ($ok) { write-host "The number is $n" -ForegroundColor cyan}
5 else { write-host 'conversion was not successful' -ForegroundColor red}
6

PS C:\Users>
PS C:\Users> [int] $n = $null # define a variable to hold the result
[string] $s = read-host 'Enter a string'
$ok = [int]::TryParse($s, [ref] $n)
if ($ok) { write-host "The number is $n" -ForegroundColor cyan}
else { write-host 'conversion was not successful' -ForegroundColor red}

Enter a string: 1024
The number is 1024

PS C:\Users> [int] $n = $null # define a variable to hold the result
[string] $s = read-host 'Enter a string'
$ok = [int]::TryParse($s, [ref] $n)
if ($ok) { write-host "The number is $n" -ForegroundColor cyan}
else { write-host 'conversion was not successful' -ForegroundColor red}
Enter a string: abc
conversion was not successful

```

[ref] generates a pointer to the variable *\$n* which is subsequently passed to TryParse. A pointer is required in this case since TryParse updates the contents of the parameter. If *[ref]* is not specified, an error is thrown since what is passed is the equivalent of a read-only value.

Regular Expressions

Tuesday, April 16, 2013
11:46 AM

A regular expression is a pattern that describes a text. Regular expressions are primarily used for searching text for matching substrings and, in some cases, for extracting substrings from text.

"There is no gentle beginning to regular expressions. You are either into hieroglyphics big time - in which case you will love this stuff - or you need to use regular expression, in which case your only reward may be a headache."

<http://www.zytrax.com/tech/web/regex.htm>

<http://msdn.microsoft.com/en-us/library/az24scfc.aspx>

<https://technet.microsoft.com/en-us/magazine/2007.11.powershell.aspx>

PowerShell by default does case insensitive match when using the *-match* operator. If case sensitive matching is required, the *-cmatch* operator should be used.

Definitions:

literal	Any character or characters used in a search or matching expression.
metacharacter	Special characters that have unique meanings, .e.g., ^,[,]
escape	The escape character \ converts a metacharacter into a literal

Metacharacters:

There are many metacharacters included in regular expressions. Below are some of the more common ones.

The dot (or period) matches exactly one character. In the examples below the string matches if it substring beginning with a "D" or "d", followed by one character, followed by "n" is found anywhere in the string.

```
PS C:\Users>
PS C:\Users> 'Don' -match 'D.n'
True
PS C:\Users> 'Dan' -match 'D.n'
True
PS C:\Users> 'dun' -match 'D.n'
True
PS C:\Users> 'dun' -cmatch 'D.n'
False
PS C:\Users> 'Done' -match 'D.n'
True
PS C:\Users> 'Done with it' -match 'D.n'
True
PS C:\Users> 'Doone' -match 'D.n'
False
PS C:\Users> 'Dne' -match 'D.n'
False
PS C:\Users>
```

The question mark ? matches 0 or 1 instance of a character. In this example, if a substring that begins with "D" or "d", followed by zero or one character, followed by an "n" is found the string than the match occurs.

```
[regex] $r = 'D?n'
```

```
PS C:\Users>
PS C:\Users> $r = 'D?n'
PS C:\Users>
PS C:\Users> [regex] $r = 'D?n'
PS C:\Users>
PS C:\Users> 'Dan' -match $r
True
PS C:\Users> 'Don' -match $r
True
PS C:\Users> 'Dn' -match $r
True
PS C:\Users>
```

The asterisk * tells the regular expression engine to match the current character 0 or more times.

```
[regex] $r = 'D*n'
```

```
PS C:\Users>
PS C:\Users> [regex] $r = 'D*n'
PS C:\Users>
PS C:\Users> 'Dan' -match $r
True
PS C:\Users> 'Don' -match $r
True
PS C:\Users> 'Doone' -match $r
True
PS C:\Users>
```

The [|] allows matching multiple characters in the position. In the example below, the first character of a matching substring must be a "D" or "d" followed by an "a", "o", "I", or "u" or their upper case equivalents followed by an "n" or "N".

```
[regex] $r = 'D[a|o|i|u]n'
```

```
PS C:\Users>
PS C:\Users> [regex] $r = 'D[a|o|i|u]n'
PS C:\Users>
PS C:\Users> 'Dan' -match $r
True
PS C:\Users> 'Don' -match $r
True
PS C:\Users> 'Den' -match $r
False
PS C:\Users> 'Dun' -match $r
True
```

The repeater $\{n\}$ (where n is a number) allows the element to be matched exactly n times. In the example below, [a|o] means either the character "a" or the character "o". The regular expression below requires that the matching substring begin with "D" (or "d") followed by exactly two characters which must be a combination of "o" or "a" followed by an "n".

```
[regex] $r = 'D[a|o]{2}n'
```

```
PS C:\Users>
PS C:\Users> [regex] $r = 'D[a|o]{2}n'
PS C:\Users>
PS C:\Users> 'Dan' -match $r
False
PS C:\Users> 'Daan' -match $r
True
PS C:\Users> 'D0n' -match $r
False
PS C:\Users> 'Don' -match $r
False
PS C:\Users> 'Doon' -match $r
True
PS C:\Users> 'Doan' -match $r
True
PS C:\Users> 'Daon' -match $r
True
PS C:\Users> 'Daoan' -match $r
False
PS C:\Users> 'Daxn' -match $r
False
PS C:\Users>
```

We may also have repeater ranges. A repeater in the form $\{n,\}$ means element must match at least n times but may match more than n times. In the example below, we must have at least one occurrence of the two character combination consisting of "a" or "o". However we may have more two characters provided the character is "a" or "o".

```
[regex] $r = 'D[a|o]{2,}n'
```

```
PS C:\Users>
PS C:\Users> [regex] $r = 'D[a|o]{2,}n'
PS C:\Users>
PS C:\Users> 'Dan' -match $r
False
PS C:\Users> 'Daan' -match $r
True
PS C:\Users> 'Doon' -match $r
True
PS C:\Users> 'Doan' -match $r
True
PS C:\Users> 'Daoan' -match $r
True
PS C:\Users> 'Daoaoaoaoaon' -match $r
True
PS C:\Users> 'Daoaxn' -match $r
False
PS C:\Users>
```

Finally we can limit the number of occurrences using the syntax $\{n,m\}$. In this case the element must match at least n times but no more than m times. In the example below, the characters "a" or "o" must be repeated at least twice but not more than three times.

```
[regex] $r = 'D[a|o]{2,3}n'
```

```
PS C:\Users>
PS C:\Users> [regex] $r = 'D[a|o]{2,3}n'
PS C:\Users>
PS C:\Users> 'Don' -match $r
False
PS C:\Users> 'Doon' -match $r
True
PS C:\Users> 'Daon' -match $r
True
PS C:\Users> 'Daoan' -match $r
True
PS C:\Users> 'Daoon' -match $r
True
PS C:\Users> 'Daoooan' -match $r
False
PS C:\Users>
```

The \d indicates one digit. The expression below requires that the matching text begin with the character "a" followed by exactly two digits followed by the character "b".

```
[regex] $r = 'a\d\db'
```

```
PS C:\Users>
PS C:\Users> [regex] $r = 'a\d\db'
PS C:\Users>
PS C:\Users> 'a12b' -match $r
True
PS C:\Users> 'alb' -match $r
False
PS C:\Users> 'a78b' -match $r
True
PS C:\Users> 'a7#b' -match $r
False
PS C:\Users>
```

We may also use repeaters with digit metacharacters. The example below matches five digits. The metacharacters ".*" means match any characters zero or more times.

```
[regex] $r = '.*\d{5}.*'
```

```
PS C:\Users>
PS C:\Users> [regex] $r = '.*\d{5}.*'
PS C:\Users>
PS C:\Users> 'Ann Arbor, Michigan 48105' -match $r
True
PS C:\Users> 'Ann Arbor, Michigan ' -match $r
False
PS C:\Users> 'Ann Arbor, Michigan 48' -match $r
False
PS C:\Users> 'Ann Arbor, Michigan 4810' -match $r
False
PS C:\Users> 'Ann Arbor, Michigan 481056' -match $r
True
PS C:\Users>
```

In the above example, we see that the regular expression pattern is not sufficient to match only five digits. The regex matched five digits which was all this was specified. The fact a sixth digit was present had no effect. To match exactly five digits we need to be more restrictive with the regex. In the expression below, the "\b" indicates a word boundary; so the regular expression matches five digits on a word boundary at both ends. A word boundary means that the five digits may be preceded and succeeded by a space or punctuation.

```
[regex] $r = '\b\d{5}\b'
```

```
PS C:\Users>
PS C:\Users> [regex] $r = '\b\d{5}\b'
PS C:\Users>
PS C:\Users> ',48105 ' -match $r
True
PS C:\Users> ' 48105 ' -match $r
True
PS C:\Users> ' 48105, ' -match $r
True
PS C:\Users> ' 481056 ' -match $r
False
```

```
$s ='the quick brown fox jumps over the lazy dog'

# match any number of characters before the string lazy
$s -match '.*lazy'

# match either brown or red
$s -match 'brown|red'

$s1 ='the quick red fox jumps over the lazy dog'
$s1 -match 'brown|red'

$s1 ='the quick green fox jumps over the lazy dog'
$s1 -match 'brown|red'

# dog must appear at the end of the string
$s -match 'dog$'

# 'the' must appear at the beginning of the string
$s -match '^the'
```

```

PS C:\Users>
PS C:\Users> $s ='the quick brown fox jumps over the lazy dog'
PS C:\Users> # match any number of characters before the string lazy
PS C:\Users> $s -match '.*lazy'
True
PS C:\Users>
PS C:\Users> # match either brown or red
PS C:\Users> $s -match 'brown|red'
True
PS C:\Users>
PS C:\Users> $s1 ='the quick red fox jumps over the lazy dog'
PS C:\Users> $s1 -match 'brown|red'
True
PS C:\Users>
PS C:\Users> $s1 ='the quick green fox jumps over the lazy dog'
PS C:\Users> $s1 -match 'brown|red'
False
PS C:\Users>
PS C:\Users> # dog must appear at the end of the string
PS C:\Users> $s -match 'dog$'
True
PS C:\Users>
PS C:\Users> # 'the' must appear at the beginning of the string
PS C:\Users> $s -match '^the'
True

```

The next example validates the format of a date in the form 'year-month-day'. It does not validate the actual date, so a string may satisfy the regular expression but not be a valid date.

```
# validate a date
'2013-April-17' -match '\d{4}-[a-z]{3,9}-\d{2}'
```

```

PS C:\Users>
PS C:\Users> # validate a date
PS C:\Users>
PS C:\Users> '2013-April-17' -match '\d{4}-[a-z]{3,9}-\d{2}'
True
PS C:\Users> '2013-Abcd-17' -match '\d{4}-[a-z]{3,9}-\d{2}'
True .

```

The regular expression used is explained in the table below.

\d{4}	Must be one digit (d) which may repeat up to 4 times
-------	--

-	Followed by a -
[a-z]{3,9}	Followed by any character in the set a through z which must repeat a minimum of 3 times up to a maximum of 9
-	Followed by a -
\d{2}	Followed by a digit that must repeat up to 2 times

We may also use capture groups to parse a string into its component parts. Capture groups are indicated by enclosing part of the pattern in parentheses. When capture groups are specified, the parsed components are found in the \$matches array.

```
[regex] $r =  '(\d{4})-([A-z]{3,9})-(\d{2})'
'2013-April-17' -match $r
```

```
PS C:\Users>
PS C:\Users> [regex] $r =  '(\d{4})-([A-z]{3,9})-(\d{2})'
PS C:\Users>
PS C:\Users> '2013-April-17' -match $r
True
PS C:\Users>
PS C:\Users> $matches
```

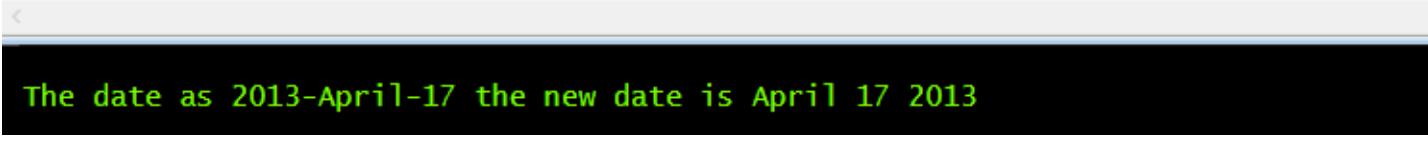
Name	Value
0	2013-April-17
1	2013
2	April
3	17

In the above example, `(\d{4})` means that the four digits matched are captured in the \$matches array, likewise for `([a-z]{3,9})` and `(\d{2})`. In the above illustration where the contents of \$matches is displayed, the name corresponds to the capture variable created for each group. This means that name 1 refers to the capture variable 1, which captured the year; name 2 refers to \$2, which is captured the month, and name 3 refers to \$3, which captured the day.

We may use the capture variables to reorder the date as shown below.

```
[regex] $r = '(\d{4})-([A-z]{3,9})-(\d{2})'
$date = '2013-April-17'
$matched = $date -match $r
if ($matched) {
    $newdate = $date -replace $r, '$2 $3 $1'
} else {
    $newdate = 'unconvertable'
}
write-host "The date as $date the new date is $newdate"
```

```
1 clear-host
2 write-host
3 [regex] $r = '(\d{4})-([A-z]{3,9})-(\d{2})'
4 $date = '2013-April-17'
5 $matched = $date -match $r
6 if ($matched) {
7     $newdate = $date -replace $r, '$2 $3 $1'
8 } else {
9     $newdate = 'unconvertable'
10 }
11 write-host "The date as $date the new date is $newdate`n"
```



```
The date as 2013-April-17 the new date is April 17 2013
```

From the above example, it should be apparent that the capture variables are simply indexes into the \$matches array:

\$1	->	\$matches[1]	2013	(\d{4})
\$2	->	\$matches[2]	April	([a-z]{3,9})
\$3	->	\$matches[3]	17	(\d{2})

Validating an IP address:

In this example, we validate the form of the IP address. It's important as is shown in the execution illustration that only the form is validate and not the IP address. The form may be valid but the IP address may not.

```
[regex] $r = '\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b'
'207.75.134.1' -match $r
```

```
'207.75.134.A' -match $r  
'207.75.134.999' -match $r
```

```
PS C:\Users>  
PS C:\Users> [regex] $r = '\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b'  
PS C:\Users>  
PS C:\Users> '207.75.134.1' -match $r  
True  
PS C:\Users>  
PS C:\Users> '207.75.134.A' -match $r  
False  
PS C:\Users>  
PS C:\Users> '207.75.134.999' -match $r  
True  
PS C:\Users>
```

As shown in the above, '`207.75.134.A`' `-match $r` fails because the last octet is not numeric. However, the IP address '`207.75.134.999`' `-match $r` matches the syntax as specified in the regular expression but semantically is not a valid IP address. The regular expression currently defined is not sufficient to validate an IP address.

One way of validating the IP address is to parse the IP address into its separate octets by adding capture groups. The function below does just that, then validates each octet.

```
function validIPAddr ( [string] $IPAddr ) {  
    [regex] $r = '\b(\d{1,3})\.( \d{1,3})\.( \d{1,3})\.( \d{1,3})\b'  
    $matched = $IPAddr -match $r  
    if ( -not $matched ) {  
        return $false  
    }  
  
    $valid = $true  
    for ($i=1; $i -le 4; $i++) {  
        # force cast of string to integer  
        [int] $octet = $matches[$i]  
        if ($octet -lt 1 -or $octet -gt 255) {  
            $valid = $false  
            break  
        }  
    }  
}
```

```
    return $valid
}
```

The script below uses that function to validate several IP addresses.

```
function validIPAddr ( [string] $IPAddr ) {
[regex] $r = '\b(\d{1,3})\.( \d{1,3})\.( \d{1,3})\.( \d{1,3})\b'
$matched = $IPAddr -match $r
if ( -not $matched ) {
    return $false
}

$valid = $true
for ($i=1; $i -le 4; $i++) {
    # for cast of string to integer
    [int] $octet = $matches[$i]
    if ($octet -lt 1 -or $octet -gt 255) {
        $valid = $false
        break
    }
}
if ( $valid ) {
    return $true
} else {
    return $false
}
}

clear-host
write-host

$addr =
'207.75.134.999','207.75.134.254','207.75.134.1','207.75.134.a'
for ($i=0; $i -lt $addr.length; $i++) {
    if ( validIPAddr $($addr[$i]) ) {
        write-host "IP address $($addr[$i]) is valid" -ForegroundColor Green
    } else {
        write-host "IP address $($addr[$i]) is not valid" -ForegroundColor Red
    }
}
```

```

1 function validIPAddr ([string] $IPAddr) {
2     [regex] $r = '\b(\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3})\b'
3     $matched = $IPAddr -match $r
4     if (-not $matched) {
5         return $false
6     }
7
8     $valid = $true
9     for ($i=1; $i -le 4; $i++) {
10        # for cast of string to integer
11        [int] $octet = $matches[$i]
12        if ($octet -lt 1 -or $octet -gt 255) {
13            $valid = $false
14            break
15        }
16    }
17    if ( $valid ) {
18        return $true
19    } else {
20        return $false
21    }
22 }
23
24 clear-host
25 write-host
26
27 $addr = '207.75.134.999', '207.75.134.254', '207.75.134.1', '207.75.134.a'
28 for ($i=0; $i -lt $addr.length; $i++) {
29     if ( validIPAddr $($addr[$i]) ) {
30         write-host "IP address $($addr[$i]) is valid" -ForegroundColor Green
31     } else {
32         write-host "IP address $($addr[$i]) is not valid" -ForegroundColor Red
33     }
34 }
```

```

IP address 207.75.134.999 is not valid
IP address 207.75.134.254 is valid
IP address 207.75.134.1 is valid
IP address 207.75.134.a is not valid
```

That above is a lot of code to solve this problem. Let's look at an alternative approach using regular expressions to validate an IP address.

```
[regex] $r = '\b((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b'
```

```
\b
( ( 25[0-5] | 2[0-4][0-9] | [01]? [0-9] [0-9]? ) \. ){3} # first three octets
( 25[0-5] | 2[0-4][0-9] | [01]? [0-9] [0-9]? )      # last octet
\b
```

\b	Begins on a word boundary
(Begin a repeat group (for three octets)
(Begin another group for one octet
25[0-5]	Match 250 through 255
	Or
2[0-4][0-9]	Match 200 through 249
	Or
[01]?	Match the digits 0 or the digit 1 zero or one time
[0-9]	Match the digits 0 through 9
[0-9]?	Match the digits 0 through 9 zero or one time
)	End the octet group
\.	Followed by a dot
)\{3}	End the repeat group and repeat it three times
(Begin the last octet group
25[0-5]	Match 250 through 255
	Or
2[0-4][0-9]	Match 200 through 249
	Or
[01]?	Match the digits 0 or the digit 1 zero or one time
[0-9]	Match the digits 0 through 9
[0-9]?	Match the digits 0 through 9 zero or one time
)	End the last octet group

The above function is rewritten to use this regular expression to validate the IP address. The regular expression is broken into two parts for the sake of readability.

```
function validIPAddr ( [string] $IPAddr ) {
    $part1 = '\b((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)\{3\}'
    $part2 = '(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b'
```

```

[regex] $r = $part1 + $part2
return ( $IPAddr -match $r )
}

```

```

1 function validIPAddr ( [string] $IPAddr ) {
2     $part1 = '\b((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}\b'
3     $part2 = '(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b'
4     [regex] $r = $part1 + $part2
5     return ( $IPAddr -match $r )
6 }
7
8 clear-host
9 write-host
10
11 $addr = '207.75.134.999', '207.75.134.254', '207.75.134.1', '207.75.134.a'
12 for ($i=0; $i -lt $addr.length; $i++) {
13     if ( validIPAddr ($addr[$i]) ) {
14         write-host "IP address $($addr[$i]) is valid" -ForegroundColor Green
15     } else {
16         write-host "IP address $($addr[$i]) is not valid" -ForegroundColor Red
17     }
18 }
19

```

```

IP address 207.75.134.999 is not valid
IP address 207.75.134.254 is valid
IP address 207.75.134.1 is valid
IP address 207.75.134.a is not valid
PS C:\>

```

Validating an email address:

```

[regex] $r = '\b[A-z0-9._%+-]+@[A-z0-9.-]+\.[A-z]{2,4}\b'
$emailaddr = "test@somewhere.com"
$emailaddr -match $r

```

\b	Begins on a word boundary
[A-z0-9._%+-]	Match A thru z (upper and lowercase alphabet), 0 through 9, and the indicated special characters
+	Match against the previous metacharacters 1 or more times
@	Followed by an @
[A-z0-9.-]	Match any of the indicated characters

+	Repeat the previous metacharacter match 1 or more times
\.	Followed by a dot
[A-z]{2,4}	Match a minimum of two letters up to a maximum of four letters

```
PS C:\Users>
PS C:\Users> [regex] $r = '\b[A-z0-9._%+-]+@[A-z0-9.-]+\.\w{2,4}\b'
PS C:\Users> $emailaddr = "test@somewhere.com"
PS C:\Users> $emailaddr -match $r
True
PS C:\Users> $emailaddr = "test&somewhere.com"
PS C:\Users> $emailaddr -match $r
False
```

Output Formatting

Monday, April 15, 2013
7:44 PM

The format operator (-f) requires formatting string on the left side and variables to be formatted on the right side. The left side of the format operator is a format string consisting of elements enclosed in braces. Each set of element encloses a field index number followed by an optional field length followed by an optional format specifier. Each element that correspond to a variable or expression on the right side of the format operator.

<https://devcentral.f5.com/articles/powershell-abcs-f-is-for-format-operator#.UnLAtfmsiG4>

The format string:

- '{0}' A formatting element that is applied to the first variable on the right side. No field length or format specifier is used.
- '{0,10}' The first variable is displayed right-aligned in field of 10 characters.

- | | |
|-------------------|--|
| '{0,-10}' | The first variable is displayed left-aligned in field of 10 characters. |
| '{0:0%} {1,5:x4}' | The first variable is displayed as a percent. Since no field length is specified enough space is allocated to accommodate the output. The second variable is display as a hexadecimal number right-aligned in a field of 5 characters. |

"{0}	{1}	{2}"	-f \$s ,	(1gb/1mb),	'a string'
"{1}	{2}	{0}"	-f \$s ,	(1gb/1mb),	'a string'
"{1}	{1}	{1}"	-f \$s ,	(1gb/1mb),	'a string'

The output field length may be specified by following the number by a comma and the field length. A negative field length means left-aligned"

"{0,-20}	{1,20}"	-f 'This is left aligned',	'This is right aligned'
----------	---------	----------------------------	-------------------------

Formatting Numbers:

Symbol	Type	Call	Result
#	Digit placeholder	"{0:(#).##}" -f \$value	(1000000)
%	Percentage	"{0:0%}" -f \$value	100000000%
,	Thousands separator	"{0:0,0}" -f \$value	1,000,000
,	Integral multiple of 1,000	"{0:0,.}" -f \$value	1000
.	Decimal point	"{0:0.0}" -f \$value	1000000.0
0	0 placeholder	"{0:00.0000}" -f \$value	1000000.0000
c	Currency	"{0:c}" -f \$value	1,000,000.00 â,¬
d	Decimal	"{0:d}" -f \$value	1000000
e	Scientific notation	"{0:e}" -f \$value	1.000000e+006
e	Exponent wildcard	"{0:00e+0}" -f \$value	10e+5
f	Fixed point	"{0:f}" -f \$value	1000000.00
g	General	"{0:g}" -f \$value	1000000
n	Thousands separator	"{0:n}" -f \$value	1,000,000.00
x	Hexadecimal	"0x{0:x4}" -f \$value	0x4240

Formatting Dates:

Symbol	Type	Call	Result
d	Short date format	"{0:d}" -f \$value	09/07/2007
D	Long date format	"{0:D}" -f \$value	Friday, September 7, 2007
t	Short time format	"{0:t}" -f \$value	10:53 AM
T	Long time format	"{0:T}" -f \$value	10:53:56 AM
f	Full date and time (short)	"{0:f}" -f \$value	Friday, September 7, 2007 10:53 AM
F	Full date and time (long)	"{0:F}" -f \$value	Friday, September 7, 2007 10:53:56 AM
g	Standard date (short)	"{0:g}" -f \$value	09/07/2007 10:53 AM
G	Standard date (long)	"{0:G}" -f \$value	09/07/2007 10:53:56 AM
M	Day of month	"{0:M}" -f \$value	September 07
r	RFC1123 date format	"{0:r}" -f \$value	Fri, 07 Sep 2007 10:53:56 GMT
s	Sortable date format	"{0:s}" -f \$value	2007-09-07T10:53:56
u	Universally sortable date format	"{0:u}" -f \$value	2007-09-07 10:53:56Z
U	Universally sortable GMT date format	"{0:U}" -f \$value	Friday, September 7, 2007 08:53:56
Y	Year/month format pattern	"{0:Y}" -f \$value	September 2007

Symbol	Type	Call	Result
dd	Day of month	"{0:dd}" -f \$value	07
ddd	Abbreviated name of day	"{0:ddd}" -f \$value	Fri
dddd	Full name of day	"{0:dddd}" -f \$value	Friday
gg	Era	"{0:gg}" -f \$value	A. D.
hh	Hours from 01 to 12	"{0:hh}" -f \$value	10
HH	Hours from 0 to 23	"{0:HH}" -f \$value	10
mm	Minute	"{0:mm}" -f \$value	53
MM	Month	"{0:MM}" -f \$value	09
MMM	Abbreviated month name	"{0:MMM}" -f \$value	Sep
MMMM	Full month name	"{0:MMMM}" -f \$value	September
ss	Second	"{0:ss}" -f \$value	56
tt	AM or PM	"{0:tt}" -f \$value	
yy	Year in two digits	"{0:yy}" -f \$value	07
yyyy	Year in four digits	"{0:YY}" -f \$value	2007
zz	Time zone including leading zero	"{0:zz}" -f \$value	+02
zzz	Time zone in hours and minutes	"{0:zzz}" -f \$value	+02:00

Introduction

Sunday, January 13, 2013
12:11 PM

[Using Windows PowerShell](#)

Piping is the process of presenting output objects from one cmdlet as input objects to another. The pipeline operator is the vertical bar (|) and connects one cmdlet to another. An example of pipeline is shown below. The get-childitem cmdlet retrieves files (-file) and creates associated *fileinfo objects*. As each object is created it is sent to the cmdlet out-host for processing.

```
get-childitem c:\windows\system32 -file | out-host -paging
```

The example above demonstrates streaming of objects through the pipeline. Piping may occur in one of two different modes, *sequential* or *streaming*:

1. *Sequential mode* collects all the objects outputted by the current cmdlet before piping them to the next cmdlet in the pipeline. This *blocking* mode may result in a significant delay before output appears in the console window.
2. *Streaming mode* immediately pipes the object from current cmdlet to the next cmdlet in the pipeline. In this mode, objects are continuously flowing through the pipeline as they are created and output appears almost immediately in the console window.

The mode of piping used is determine by the developer of the cmdlet. We can demonstrate this using the get-childitem cmdlet. In the example gci retrieves a collection of all file system objects (files and directories) in the c:\windows directory:

```
# more collects all objects before showing any properties. Notice  
the delay  
Get-ChildItem c:\windows\system32 -recurse | More
```

```
# out-host starts displaying the object properties immediately  
Get-ChildItem c:\windows\system32 -recurse | Out-Host -paging
```

As demonstrated above, Sequential processing may have undesirable effects. *Blocking* results when the cmdlet has to process a significant amount of data. The effect is a long wait before pipeline output appears. Another undesirable effect is the large memory usage which may cause Windows to page.

At the end of the end of the pipeline, selected objects are converted to text. If the end of the pipe lacks a cmdlet to convert an object to text, PowerShell will implicitly add cmdlets to do so. To convert objects to text , PowerShell implicitly pipes object output to out-default which in turn pipes the output to format-table and out-host.

```
# this  
get-childitem  
# is the equivalent of this  
get-childitem | out-default  
# which is the equivalent of this  
get-childitem | format-table | out-host
```

The cmdlets format-table and format-list may be used at the end of the pipeline to convert object properties to text and format the display. As the name implies, format-table lists the default properties to display in a horizontal tabular fashion while format-list lists them in a vertical fashion.

These cmdlets are a good way to determine the properties available at the end of a pipeline. In the syntax below, get-childitem generates fileinfo and directoryinfo objects and pipes them into format-table and format-list. Using the wildcard causes the cmdlets to display all the available properties.

```
get-childitem c:\windows | format-table *  
get-childitem c:\windows | format-list *
```

```
PS C:\user>
PS C:\user> get-childitem c:\windows | format-table *

```

PSPath	PSParentPath	PSChildName	PSDrive	PSProvider	PSIsContainer	BaseName	Mode
Micros...	Micros...	addons	C	Micros...	True	addons	d----
Micros...	Micros...	AppCompat	C	Micros...	True	AppCompat	d----
Micros...	Micros...	AppPatch	C	Micros...	True	AppPatch	d----
Micros...	Micros...	assembly	C	Micros...	True	assembly	d-r-
Micros...	Micros...	Boot	C	Micros...	True	Boot	d----
Micros...	Micros...	Branding	C	Micros...	True	Branding	d----
Micros...	Micros...	CSC	C	Micros...	True	CSC	d----
Micros...	Micros...	Cursors	C	Micros...	True	Cursors	d----
Micros...	Micros...	debug	C	Micros...	True	debug	d----
Micros...	Micros...	Dell	C	Micros...	True	Dell	d----
Micros...	Micros...	diagon...	C	Micros...	True	diagon...	d----
Micros...	Micros...	Digital...	C	Micros...	True	Digital...	d----
Micros...	Micros...	Downlo...	C	Micros...	True	Downlo...	d----
Micros...	Micros...	Downlo...	C	Micros...	True	Downlo...	d----

```
PS C:\user>
PS C:\user> get-childitem c:\windows | format-list *
```

PSPath	:	Microsoft.PowerShell.Core\FileSystem::C:\windows\addons
PSParentPath	:	Microsoft.PowerShell.Core\FileSystem::C:\windows
PSChildName	:	addons
PSDrive	:	C
PSProvider	:	Microsoft.PowerShell.Core\FileSystem
PSIsContainer	:	True
BaseName	:	addons
Mode	:	d----
Name	:	addons
Parent	:	windows
Exists	:	True
Root	:	C:\
FullName	:	C:\windows\addons
Extension	:	
CreationTime	:	7/14/2009 1:32:38 AM
CreationTimeUtc	:	7/14/2009 5:32:38 AM
LastAccessTime	:	7/14/2009 1:32:39 AM
LastAccessTimeUtc	:	7/14/2009 5:32:39 AM
LastWriteTime	:	7/14/2009 1:32:39 AM

As we see in the above illustrations, using format-table with a wildcard doesn't produce very useful results. The format-list is more appropriate in this case. Using a wildcard is useful in determining the properties available at the end of the pipeline. However, since many times we want to display selected properties, both allow the selection of specific properties. In the first example, we display the file name and file size (length). We also use the -autosize switch parameter to automatically adjust the spacing between columns.

```
get-childitem c:\windows | format-table name, length -AutoSize  
get-childitem c:\windows | format-list name, length
```

```
PS C:\user>  
PS C:\user> get-childitem c:\windows -file | format-table name, length -AutoSize  


| Name                 | Length  |
|----------------------|---------|
| atiogl.xml           | 18333   |
| ativpsrm.bin         | 0       |
| bfsvc.exe            | 71168   |
| bootstat.dat         | 67584   |
| csup.txt             | 12      |
| DfsrAdmin.exe        | 258048  |
| DfsrAdmin.exe.config | 1315    |
| DirectX.log          | 32161   |
| DPINST.LOG           | 12552   |
| DtcInstall.log       | 2027    |
| eplauncher.mif       | 1945    |
| explorer.exe         | 2871808 |
| fveupdate.exe        | 15360   |
| HelpPane.exe         | 733696  |
| hh.exe               | 16896   |
| IE10_main.log        | 6258    |
| IE11_main.log        | 7582    |


```

```
PS C:\user>  
PS C:\user> get-childitem c:\windows -file | format-list name, length  
  
Name : atiogl.xml  
Length : 18333  
  
Name : ativpsrm.bin  
Length : 0  
  
Name : bfsvc.exe  
Length : 71168  
  
Name : bootstat.dat  
Length : 67584  
  
Name : csup.txt  
Length : 12  
  
Name : DfsrAdmin.exe  
Length : 258048  
  
Name : DfsrAdmin.exe.config  
Length : 1315
```

Select-Object

Wednesday, September 11, 2013
12:51 PM

The select-object cmdlet is used inside the pipeline to restrict the number of objects flowing to the next cmdlet in the pipeline or to select specific properties to propagate through the pipeline. This cmdlet may also be used to create *derived* properties.

In this first example, we use the select-object to select the first five objects and pass them on through the pipeline. Any object not selected is discarded.

```
get-childItem c:\windows | select-object -first 5
```

```
PS C:\user>
PS C:\user> get-childItem c:\windows | select-object -first 5

Directory: C:\windows

Mode                LastWriteTime     Length Name
----                -----          ---- 
d----        7/14/2009    1:32 AM      addins
d----        4/10/2015   3:02 PM      AppCompat
d----        7/15/2015   3:43 AM      AppPatch
d-r-s        6/26/2015   6:33 AM      assembly
d----        7/14/2009    1:32 AM      Boot
```

In this example, we select the file or directory name of the file system object and its creation time. When selecting specific properties, select-object creates a new of the same type as the object in the pipeline. It assigns the new object the selected properties and pass the new object down the pipeline while discarding the original object.

```
get-childItem c:\windows | select-object name, creationtime | format-table -autosize
```

```
PS C:\user>
PS C:\user> get-childItem c:\windows | select-object name, creationtime | format-table -auto
Name                               CreationTime
----                               -----
addons                            7/14/2009 1:32:38 AM
AppCompat                         7/13/2009 11:20:08 PM
AppPatch                           7/13/2009 11:20:08 PM
assembly                           7/13/2009 11:20:08 PM
Boot                               7/13/2009 11:20:09 PM
Branding                           7/13/2009 11:20:09 PM
CSC                                7/14/2009 3:46:31 AM
 Cursors                            7/13/2009 11:20:09 PM
debug                               7/14/2009 12:45:54 AM
DEll                               8/16/2010 4:35:59 PM
 diagnostics                         7/14/2009 1:32:38 AM
DigitalLocker                        7/14/2009 1:37:46 AM
Downloaded Installations           8/16/2010 4:39:39 PM
Downloaded Program Files            7/14/2009 1:32:38 AM
ehome                               7/14/2009 3:46:36 AM
```

Example:

Given a csv file shown below derived from a firewall log with thousands of entries. You need to determine the unique source IP addresses represented in the file.

Date	Time	Syslog_ID	Source_IP	Source_Port	Dest_IP	Dest_Port
Jan 16 2014	14:54:31	106023	188.26.62.236	30870	207.75.134.154	8832
Jan 16 2014	14:54:31	106023	193.110.18.33	62629	207.75.134.154	8832
Jan 16 2014	14:54:30	106023	187.13.238.91	22350	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	46.249.171.169	27376	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	188.134.40.93	49611	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	187.13.238.91	13757	207.75.134.154	8832

Jan 16 2014	14:54:29	106023	80.99.73.186	16008	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	187.13.238.91	48884	207.75.134.154	8832

```
import-csv fwlog.csv | select-object Source_IP -unique
```

```
PS C:\Users>
PS C:\Users> import-csv fwlog.csv | select-object Source_IP -unique
Source_IP
-----
188.26.62.236
193.110.18.33
187.13.238.91
46.249.171.169
188.134.40.93
80.99.73.186
```

The import-csv cmdlet imports the csv file and creates custom objects with the properties corresponding to the columns in the csv file. The cmdlet select-object select creates a new object with the source_ip property and eliminates any duplicate objects, i.e., IP addresses, using the -unique switch

To confirm the properties of the imported csv file, pipe the result into get-member.

```
import-csv fwlog.csv | gm
```

```

PS C:\Users>
PS C:\Users> import-csv fwlog.csv | gm

    TypeName: System.Management.Automation.PSCustomObject

Name      MemberType   Definition
----      -----      -----
Equals    Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType   Method      type GetType()
ToString  Method      string ToString()
Date      NoteProperty System.String Date=Jan 16 2014
Dest_IP   NoteProperty System.String Dest_IP=207.75.134.154
Dest_Port NoteProperty System.String Dest_Port=8832
Source_IP NoteProperty System.String Source_IP=188.26.62.236
Source_Port NoteProperty System.String Source_Port=30870
Syslog_ID NoteProperty System.String Syslog_ID=106023
Time      NoteProperty System.String Time=14:54:31

```

Windows Management Instrumentation is the infrastructure for managing operations in a Windows network. WMI will be covered in a later section. The examples below demonstrate using the get-wmiobject cmdlet with the where-object cmdlet. One of the components of WMI are *classes* that provide information about the object being managed. The cmdlet below queries the WMI repository for information about the local user accounts. We export this information to a csv file for use with subsequent cmdlets.

```

get-wmiobject win32_useraccount | select-object name, domain |
export-csv users.csv

```

```

PS C:\Users>
PS C:\Users> get-wmiobject win32_useraccount | select-object name, domain | export-csv users.csv
PS C:\Users> get-content .\users.csv
#TYPE Selected.System.Management.ManagementObject
"name", "domain"
"Administrator", "mink"
"ftpuser", "mink"
"Guest", "mink"
"mgalea", "mink"
PS C:\Users>

```

The gwmi cmdlet queries the WMI using the win32_useraccount class. This returns a series of wmi objects with all the user properties. The select-object cmdlet selects the user name and domain properties. If the host is not connected to a domain, the domain property has the host name otherwise it contains the

domain name. We then export these two properties to a csv file called users.csv which contains:

We may now use select-object to display either the first or last objects.

```
import-csv .\users.csv | select -first 3 | format-table -autosize  
import-csv .\users.csv | select -last 2 | format-table -autosize
```

```
PS C:\Users>  
PS C:\Users> import-csv .\users.csv | select -first 3 | format-table -a  
  
name      domain  
----      -----  
Administrator mink  
ftpuser    mink  
Guest      mink  
  
PS C:\Users> import-csv .\users.csv | select -last 2 | format-table -a  
  
name      domain  
----      -----  
Guest    mink  
mgalea  mink
```

We may also use the -unique switch to retrieve unique occurrences of fields in a file.

```
import-csv .\users.csv | select domain -unique
```

```
PS C:\Users>  
PS C:\Users> get-content .\users.csv  
#TYPE Selected.System.Management.ManagementObject  
"name","domain"  
"Administrator","mink"  
"ftpuser","mink"  
"Guest","mink"  
"mgalea","mink"  
PS C:\Users> import-csv .\users.csv | select domain -unique  
  
domain  
-----  
mink
```

Creating Derived Properties

As seen in an earlier section, a derived property may be created by select-object using a hash table. The example below illustrates how a derived property may be created.

The *get-process* cmdlet returns the information about the processor time used by a process (actually all threads of a process). The TotalProcessorTime, as the name implies, provides the total processor time used by the process. The UserProcessorTime property returns the amount of time included in TotalProcessorTime spent in user mode by the process.

User mode is a state in which a user process runs has restricted access to system resources. Sometimes the process needs to run in privileged mode to execute system functions. These states are implemented to protect one process from another and to protect the system from crashes due to bugs in software.

In the example below, we use a hash table to accomplish two goals: 1) rename the UserProcessorTime property to just UserTime; 2) Create a new derived property PrivilegedTime which is calculated by subtracting UserProcessTime from TotalProcessorTime.

```
get-process | where-object { $_.TotalProcessorTime -gt 0} |  
    select-object name, totalprocessortime,  
        @{name='UserTime';expression={$_._UserProcessorTime}},  
        @{name='PrivilegedTime';expression={$_._TotalProcessorTime -  
$_._UserProcessorTIme}}
```

```

PS D:\>
PS D:\> get-process | where-object { $_.TotalProcessorTime -gt 0} |
>>   select-object name, totalprocessortime,
>>     @{name='UserTime';expression={$_.UserProcessorTime}},
>>     @{name='PrivilegedTime';expression={$_.TotalProcessorTime - $_.UserProcessorTime}}

```

Name	TotalProcessorTime	UserTime	PrivilegedTime
AcroRd32	00:00:03.9375000	00:00:02.6875000	00:00:01.2500000
AcroRd32	00:00:00.6718750	00:00:00.0937500	00:00:00.5781250
ApplicationFrameHost	00:00:59.5625000	00:00:17.7031250	00:00:41.8593750
AppVShNotify	00:00:00.0625000	00:00:00.0312500	00:00:00.0312500
audiogd	00:00:17.4218750	00:00:03.1875000	00:00:14.2343750
browserBroker	00:00:01.6406250	00:00:00.3750000	00:00:01.2656250
chrome	00:00:15.4843750	00:00:14.1250000	00:00:01.3593750
chrome	00:00:30.9843750	00:00:27.9062500	00:00:03.0781250
chrome	00:00:15.9218750	00:00:15	00:00:00.9218750
chrome	00:00:07.6718750	00:00:06.7031250	00:00:00.9687500
chrome	00:00:01.6406250	00:00:01.4062500	00:00:00.2343750
chrome	00:00:01.0468750	00:00:00.7968750	00:00:00.2500000
chrome	00:00:01.9687500	00:00:01.6718750	00:00:00.2968750
chrome	00:00:01.1250000	00:00:00.9218750	00:00:00.2031250

We may assigned the objects emitted by a pipeline to a variable. In the example below the variable \$procs is typed the same as the process object emitted by get-process. This is as expected since select-object types the new object created based upon the type of the object that it acted upon. The GetType method shows the \$procs is an array. The length property indicates that there are 66 objects in the \$procs array.

```
$procs = get-process | where-object { $_.TotalProcessorTime -gt 0} |
    select-object name, totalprocessortime,
        @{name='UserTime';expression={$_.UserProcessorTime}},
        @{name='PrivilegedTime';expression={$_.TotalProcessorTime -
$_.UserProcessorTime}}
```

```

PS C:\Users>
PS C:\Users> $procs = get-process | where-object { $_.TotalProcessorTime -gt 0 } |
>>>   select-object name, totalprocessorTime,
>>>     @{name='UserTime';expression={$_.UserProcessorTime}},
>>>     @{name='PrivilegedTime';expression={$_.TotalProcessorTime - $_.UserProcessorTime}}
PS C:\Users>
PS C:\Users> $procs | gm

TypeName: Selected.System.Diagnostics.Process

Name          MemberType    Definition
----          -----        -----
Equals        Method       bool Equals(System.Object obj)
GetHashCode   Method       int GetHashCode()
GetType       Method       type GetType()
ToString      Method       string ToString()
Name          NoteProperty string Name=CCC
PrivilegedTime NoteProperty System.TimeSpan PrivilegedTime=00:00:01.0296066
TotalProcessorTime NoteProperty timespan TotalProcessorTime=00:00:16.9729088
UserTime      NoteProperty System.TimeSpan UserTime=00:00:15.9433022

PS C:\Users> $procs | select-object -first 5

Name  TotalProcessorTime UserTime           PrivilegedTime
----  -----              -----            -----
CCC   00:00:16.9729088   00:00:15.9433022  00:00:01.0296066
chrome 00:00:52.0107334  00:00:39.6866544  00:00:12.3240790
chrome 00:00:03.8844249  00:00:03.6192232  00:00:00.2652017
chrome 00:00:00.7956051  00:00:00.7332047  00:00:00.0624004
chrome 00:00:02.7144174  00:00:02.5740165  00:00:00.1404009

PS C:\Users> $procs.GetType()

IsPublic IsSerial Name          BaseType
-----  -----   -----          -----
True    True    Object[]       System.Array

PS C:\Users> $procs[5]

Name  TotalProcessorTime UserTime           PrivilegedTime
----  -----              -----            -----
chrome 00:00:11.7624754  00:00:11.3100725  00:00:00.4524029

PS C:\Users> $procs.Length
66
PS C:\Users>

```

Where-Object

Friday, July 17, 2015
11:24 AM

<http://technet.microsoft.com/en-us/library/hh847759.aspx>

Many times we want to filter objects as they flow through the pipeline. The Where-Object cmdlet allows the user to select specific objects based upon object properties. Objects selected are output by where-object to the next cmdlet in the pipeline. Objects not selected are discarded.

The comparison operators that may be used are described in the table below. Further information is found by using *help about_comparison_operators*.

Operator	Description
-eq	Equal to
-lt	Less than
-gt	Greater than
-ge	Greater than or Equal to
-le	Less than or equal to
-ne	Not equal to
-like	Comparision to pattern with wildcards
-notlike	Opposite of -like
-match	Regular expression comparison
-notmatch	Opposite of -match
-contains	Returns true if collection contains an object
-notcontains	Opposite of -contains
-in	Identical to -contains but the operands are reversed
-notin	Opposite of -in

The examples below demonstrate using where-object to select only those file system objects that have an extension of ".exe". The three different examples produce the same results using different comparison operators. The get-childitem switch -recurse causes the cmdlet to *walk the directory tree*, i.e., navigate down into sub-directories. The \$_ referenced in the where-object cmdlet refers to the current object in the pipeline. Note that where-object may be abbreviated to *where* or replaced by its alias the *question mark* (?).

```

# Check that the Extension property of the file system object is
equal to .exe
Get-ChildItem c:\windows\system32 -recurse |
    where-object { $_.Extension -eq '.exe' }

# Check that the Name property has a value ends with .exe.
# The -like operator allows using wildcards.
gci c:\windows\system32 -recurse | where { $_.Name -like '*.exe' }

# Use the -match operator to compare against regular expression
gci c:\windows\system32 -recurse | ? { $_.Name -match '.*exe$' }

```

The braces used in the `where-object` delimit a *scriptblock*. In this case the scriptblock consists of just one *conditional expression*; but it may be a number of PowerShell statements that return a true or false value.

In the first example above, the expression `$_.Extension -eq 'exe'` is used to compare the extension of the current filesystem object in the pipeline with the value `'.exe'`. If the expression returns a true, meaning the extension matches, the object is selected and is passed through the pipeline otherwise the object is dropped.

The second example accomplishes the same result as the first example but in this case the `-like` operation indicates that a wildcard match is to occur. The expression `$_.Name -like '*.exe'` matches the name of the filesystem object to the wildcard pattern, interpreted as any number of characters followed by `.exe`. If the Name matches the object is passed through the pipeline.

The final example uses the `-match` operator resulting in a regular expression comparison to occur. The regular expression `'.*exe$'` is interpreted as any number of character `(.)` followed by `exe` anchored at the end, i.e., the `exe` occurs at the end of the Name.

The `Where-Object` cmdlet is used so often that the "?" alias was created to minimize typing.

<http://technet.microsoft.com/en-us/library/ee177028.aspx>

Conditional expressions available in PowerShell allow the script writer to test for certain conditions and take alternative actions as a the result of the test. A

conditional expression returns a result of true or false. Because we have conditional expression that return true or false values and because we have *boolean* property types which have a true or false value, PowerShell implements the built-in variables \$true and \$false for use in comparisons.

Before exploring conditional expressions further, let's look at the properties of file system objects by inspecting the properties of the files present in c:\windows\fonts. In the pipeline below, we use get-childitem to return the fileInfo objects corresponding to the files in this directory. For the sake of convenience, we reduce the amount displayed by using the select-object cmdlet to select the first object and discard the rest. That object is piped to get-member where we specify the MemberType parameter to select only properties. The property of interest is IsReadOnly. The type of that property is bool which is short for *boolean* meaning that it may have a value or true or false. Obviously if that property is true, the file is read-only, otherwise the file is read/write.

```
PS C:\Users>
PS C:\Users> get-childitem c:\windows\fonts | select-object -first 1 | get-member -MemberType Properties

TypeName: System.IO.FileInfo
Name          MemberType      Definition
----          --          --
Mode          CodeProperty   System.String Mode{get=Mode;}
PSChildName  NoteProperty   System.String PSChildName=8514fix.fon
PSDrive       NoteProperty   System.Management.Automation.PSDriveInfo PSDrive=C
PSIsContainer NoteProperty   System.Boolean PSIsContainer=False
PSParentPath  NoteProperty   System.String PSParentPath=Microsoft.PowerShell.Core\FileSystem::C:\wind
PSPath        NoteProperty   System.String PSPath=Microsoft.PowerShell.Core\FileSystem::C:\windows\fo
PSProvider    NoteProperty   System.Management.Automation.ProviderInfo PSProvider=Microsoft.PowerShel
Attributes    Property       System.IO.FileAttributes Attributes {get;set;}
CreationTime  Property       datetime CreationTime {get;set;}
CreationTimeUtc Property     datetime CreationTimeUtc {get;set;}
Directory    Property       System.IO.DirectoryInfo Directory {get;}
DirectoryName Property     string DirectoryName {get;}
Exists        Property     bool Exists {get;}
Extension     Property     string Extension {get;}
FullName      Property     string FullName {get;}
IsReadOnly    Property     bool IsReadOnly {get;set;}
LastAccessTime Property     datetime LastAccessTime {get;set;}
LastAccessTimeUtc Property   datetime LastAccessTimeUtc {get;set;}
```

This example illustrates how we may use conditional expressions and where-object to select only read-only files. The conditional expression `$_._Readonly -eq $true` returns true or false. As we know, if the conditional expression in the scriptblock returns true, where-object passes the current object down the pipeline. If false, where-object discards the current object.

```
get-childitem c:\windows\fonts | select-object -first 5  
get-childitem c:\windows\fonts | where-object { $_.IsReadOnly -eq  
$true}
```

```
PS C:\Users>  
PS C:\Users> get-childitem c:\windows\fonts | select-object -first 5  
  
Directory: C:\windows\fonts  
  
Mode                LastWriteTime      Length Name  
----                -----          ----  --  
-a---        6/10/2009  4:43 PM       10976 8514fix.fon  
-a---        6/10/2009  4:43 PM       10976 8514fixe.fon  
-a---        6/10/2009  4:43 PM      11520 8514fixg.fon  
-a---        6/10/2009  4:43 PM       10976 8514fixr.fon  
-a---        6/10/2009  4:43 PM      11488 8514fixt.fon  
  
PS C:\Users> get-childitem c:\windows\fonts | where-object { $_.IsReadOnly -eq $true}  
  
Directory: C:\windows\fonts  
  
Mode                LastWriteTime      Length Name  
----                -----          ----  --  
-ar--        12/7/2010  2:19 PM       7276  TSPECIAL1.TTF
```

Since the where-object is expecting a result of true or false from the scriptblock and the possible values for the property IsReadOnly are true or false, we may reduce the above pipeline.

```
get-childitem c:\windows | where-object { $_.IsReadOnly }
```

The followings lists more examples of using where-object using simple conditional expressions.

```
# select files whose size is greater than or equal to 10MB  
get-childitem c:\windows\fonts -file | where-object { $_.length -ge  
10MB }
```

```

PS C:\Users> get-childitem c:\windows\fonts -file | where-object { $_.length -ge 10MB }

Directory: C:\windows\fonts

Mode                LastWriteTime      Length Name
----              -----          ----  --
-a---       11/18/2002  5:44 PM    23275812 ARIALUNI.TTF
-a---       6/10/2009   4:43 PM    16264732 batang.ttc
-a---       6/10/2009   4:43 PM    13524972 gulim.ttc
-a---       6/10/2009   4:43 PM    32217124 mingliu.ttc
-a---       6/10/2009   4:43 PM    33805700 mingliub.ttc
-a---     2/13/2015   2:51 PM    21302624 MSJH.TTC
-a---       6/10/2009   4:43 PM    21663376 msjh.ttf
-a---     2/13/2015   2:51 PM    14343024 MSJHBD.TTC
-a---       6/10/2009   4:43 PM    14512072 msjhbd.ttf
-a---     2/13/2015   2:51 PM    21543568 MSYH.TTC
-a---       6/10/2009   4:43 PM    21767952 msyh.ttf
-a---     2/13/2015   2:51 PM    14381616 MSYHBD.TTC
-a---       6/10/2009   4:43 PM    14602860 msyhbd.ttf
-a---       6/10/2009   4:43 PM    10576012 simfang.ttf
-a---       6/10/2009   4:43 PM    11785184 simkai.ttf
-a---       6/10/2009   4:43 PM    15323200 simsun.ttc
-a---       6/10/2009   4:43 PM    15406288 simsunb.ttf

PS C:\Users> get-childitem c:\windows\fonts -file | where-object { $_.length -ge 20MB }

Directory: C:\windows\fonts

Mode                LastWriteTime      Length Name
----              -----          ----  --
-a---       11/18/2002  5:44 PM    23275812 ARIALUNI.TTF
-a---       6/10/2009   4:43 PM    32217124 mingliu.ttc
-a---       6/10/2009   4:43 PM    33805700 mingliub.ttc
-a---     2/13/2015   2:51 PM    21302624 MSJH.TTC
-a---       6/10/2009   4:43 PM    21663376 msjh.ttf
-a---     2/13/2015   2:51 PM    21543568 MSYH.TTC
-a---       6/10/2009   4:43 PM    21767952 msyh.ttf

```

```

# see which services are stopped
get-service | where-object { $_.status -eq 'stopped' }

```

```

PS C:\Users>
PS C:\Users> get-service | where-object { $_.status -eq 'stopped' }

Status    Name          DisplayName
----      --          -----
Stopped  AdobeFlashPlaye... Adobe Flash Player Update Service
Stopped  AeLookupSvc    Application Experience
Stopped  ALG           Application Layer Gateway Service
Stopped  AppIDSvc     Application Identity
Stopped  AppMgmt       Application Management
Stopped  aspnet_state   ASP.NET State Service
Stopped  atashost      WebEx Service Host for Support Center
Stopped  AxInstSV     ActiveX Installer (AxInstSV)
Stopped  BBSvc         BingBar Service
Stopped  BDESVC        BitLocker Drive Encryption Service
Stopped  bthserv       Bluetooth Support Service
Stopped  chromoting    Chrome Remote Desktop Service
Stopped  clr_optimizatio... Microsoft .NET Framework NGEN v2.0....
Stopped  clr_optimizatio... Microsoft .NET Framework NGEN v2.0....
Stopped  clr_optimizatio... Microsoft .NET Framework NGEN v4.0....
Stopped  clr_optimizatio... Microsoft .NET Framework NGEN v4.0....
Stopped  COMEventApp   COM+ System Application

```

```

# see which processes are using large amounts of virtual memory
get-process | where-object { $_.virtualmemoriesize64 -gt 700mb }

```

```

PS C:\Users>
PS C:\Users> get-process | where-object { $_.virtualmemoriesize64 -gt 700mb }

Handles  NPM(K)    PM(K)      WS(K)  VM(M)      CPU(s)  Id  ProcessName
----      ----      ----      ----  ----      ----  --
  5139     193    117596    202824    709     84.97  4780 OUTLOOK
    540      62    149664    164076    918      3.68  4904 powershell_ise
    579      63    148060    168452    910      3.82  4920 powershell_ise

```

```

# see all errors in the System event log
get-eventlog system | where-object { $_.EntryType -eq 'Error' }

```

```

PS C:\windows>
PS C:\windows> get-eventlog system | where-object { $_.EntryType -eq 'Error' }

  Index Time          EntryType   Source           InstanceID Message
  ----- ----          -----   -----
466287 Jul 16 04:00    Error Microsoft Antimal...      2001 %Microsoft Antimalware ha...
466102 Jul 15 03:49    Error Service Control M... 3221232472 The OracleoraclientMTSRec...
466088 Jul 15 03:49    Error Service Control M... 3221232473 The NTRU TSS v1.2.1.29 TC...
466051 Jul 15 03:49    Error atikmdag            3221268501 Display is not active
466050 Jul 15 03:49    Error atikmdag            3221277708 CPLIB :: General - Invalid ...
465970 Jul 15 03:47    Error Service Control M... 3221232472 The OracleoraclientMTSRec...
465957 Jul 15 03:47    Error Service Control M... 3221232473 The NTRU TSS v1.2.1.29 TC...
465920 Jul 15 03:46    Error atikmdag            3221268501 Display is not active
465919 Jul 15 03:46    Error atikmdag            3221277708 CPLIB :: General - Invalid ...
465647 Jul 14 10:48    Error atikmdag            3221268501 Display is not active
465606 Jul 13 22:13    Error TermDD              3221880888 The Terminal Server secur...
465585 Jul 13 21:13    Error UmrdpService        1111 Driver Microsoft XPS Docu...
465584 Jul 13 21:13    Error UmrdpService        1111 Driver Journal Note Write...
465583 Jul 13 21:13    Error UmrdpService        1111 Driver Fax - HP ENVY 7640...
465581 Jul 13 21:13    Error UmrdpService        1111 Driver HP ENVY 7640 serie...
465544 Jul 13 10:41    Error DCOM               3221235481 The description for Event...
465543 Jul 13 10:41    Error DCOM               3221235481 The description for Event...
465542 Jul 13 10:40    Error DCOM               3221235481 The description for Event...
465541 Jul 13 10:40    Error DCOM               3221235481 The description for Event...

```

Complex conditional expressions:

The criteria for selecting objects flowing through the pipeline may be more complex than comparing on a single property. PowerShell provides the logical operators shown below that allowing the script writer to build complex conditional expression for selecting objects.

Operator	Description
-not	Negates a condition
!	Not (called the bang character or just bang)
-and	All conditions must be true for the complex conditional to be true
-or	One of the conditions must be true for the complex condition to be true

The first example demonstrates the use of -not. If we want a list of read/write files the most obvious syntax would be:

```
get-childitem c:\windows\fonts | where-object { $_.IsReadOnly -eq
```

```
$false}
```

An alternative is to use the -not operator. The read-only the property IsReadyonly has a value of true or false based upon the readonly file attribute. The -not operator negates the value, so if the value is false, i.e., read/write, the value returned to where-object is true causing all read/write file objects to get selected.

```
get-childitem c:\windows\fonts | where-object { -not $_.IsReadyonly }
```

```
PS C:\Users>
PS C:\Users> get-childitem c:\windows\fonts | where-object { -not $_.IsReadyonly }

Directory: C:\windows\fonts

Mode                LastWriteTime      Length Name
----                -----          ---- 
-a---       6/10/2009 4:43 PM        10976 8514fix.fon
-a---       6/10/2009 4:43 PM        10976 8514fixe.fon
-a---       6/10/2009 4:43 PM       11520 8514fixg.fon
-a---       6/10/2009 4:43 PM       10976 8514fixr.fon
-a---       6/10/2009 4:43 PM       11488 8514fixt.fon
-a---       6/10/2009 4:43 PM       12288 8514oem.fon
-a---       6/10/2009 4:43 PM       13248 8514oeme.fon
-a---       6/10/2009 4:43 PM       12800 8514oemg.fon
-a---       6/10/2009 4:43 PM       13200 8514oemr.fon
-a---       6/10/2009 4:43 PM       12720 8514oemt.fon
-a---       6/10/2009 4:43 PM        9280 8514sys.fon
-a---       6/10/2009 4:43 PM        9504 8514syse.fon
-a---       6/10/2009 4:43 PM        9856 8514sysg.fon
-a---       6/10/2009 4:43 PM       10064 8514sysr.fon
-a---       6/10/2009 4:43 PM        9792 8514syst.fon
-a---       6/10/2009 4:43 PM       12304 85775.fon
-a---       6/10/2009 4:43 PM       12256 85855.fon
-a---       6/10/2009 4:43 PM       12336 85f1255.fon
-a---       6/10/2009 4:43 PM       12384 85f1256.fon
```

Using -and:

Let's look at some more complex conditional expressions that may be used with where-object. The result of the -and operator is shown in the table below. The is operator requires two values or expressions that evaluate to a true or false. You may however join multiple values and expressions through the successive use of -

and. The table below shows that when joining conditional expressions with -and all conditional expression must evaluate to true for the complex conditional expression to be true.

AND	Value of condition ₁ : False	Value of condition _n : True
Value of condition ₁ : False	Result of cond ₁ AND Result of cond ₂ : False	Result of cond _n AND Result of cond ₁ : False
Value of condition _n : True	Result of cond ₁ AND Result of cond _n : False	Result of cond _n AND Result of cond _n : True

The examples below use the built-in boolean variables \$true and \$false to show the operation of -and. In the first and second examples all expressions are true so the complete expression evaluates to true. In the last example, the first three expression are true but the last expression is false so the complete expression evaluates to false.

```
PS C:\Users>
PS C:\Users> $true -and $true
True
PS C:\Users> $true -and $true -and $true
True
PS C:\Users> $true -and $true -and $true -and $false
False
```

Let's look at an example were two conditions must be met for the object to be selected. We use the get-process cmdlet to return all process objects and get-member to see the properties and methods available with a process object. To minimize the amount of information returned, we use the -membertype switch is used to select only a property or a scriptproperty.

The properties of interest are processname, a string property, and CPU, a scriptproperty . A scriptproperty is a *derived* or *synthetic* property created at the time the property is queried.

Inspecting the script block associated with the scriptproperty CPU, we see that the value returned is TotalProcessorTime.TotalSeconds which means the total process time for the process represented as seconds.

```
get-process | get-member -Membertype property, scriptproperty
```

TypeName: System.Diagnostics.Process		
Name	MemberType	Definition
BasePriority	Property	int BasePriority {get;}
Container	Property	System.ComponentModel.IContainer Container {get;}
EnableRaisingEvents	Property	bool EnableRaisingEvents {get;set;}
ExitCode	Property	int ExitCode {get;}
ExitTime	Property	datetime ExitTime {get;}
Handle	Property	System.IntPtr Handle {get;}
HandleCount	Property	int HandleCount {get;}
HasExited	Property	bool HasExited {get;}
Id	Property	int Id {get;}
MachineName	Property	string MachineName {get;}
MainModule	Property	System.Diagnostics.ProcessModule MainModule {get;}
MainWindowHandle	Property	System.IntPtr MainWindowHandle {get;}
MainWindowTitle	Property	string MainWindowTitle {get;}
PeakWorkingSet	Property	int PeakWorkingSet {get;}
PeakWorkingSet64	Property	long PeakWorkingSet64 {get;}
PriorityBoostEnabled	Property	bool PriorityBoostEnabled {get;set;}
PriorityClass	Property	System.Diagnostics.ProcessPriorityClass PriorityClass {get;}
PrivateMemorySize	Property	int PrivateMemorySize {get;}
PrivateMemorySize64	Property	long PrivateMemorySize64 {get;}
PrivilegedProcessorTime	Property	timespan PrivilegedProcessorTime {get;}
ProcessName	Property	string ProcessName {get;}
ProcessorAffinity	Property	System.IntPtr ProcessorAffinity {get;set;}
Responding	Property	bool Responding {get;}
SessionId	Property	int SessionId {get;}
Site	Property	System.ComponentModel.ISite Site {get;set;}
StandardError	Property	System.IO.StreamReader StandardError {get;}
StandardInput	Property	System.IO.StreamWriter StandardInput {get;}
StandardOutput	Property	System.IO.StreamReader StandardOutput {get;}
StartInfo	Property	System.Diagnostics.ProcessStartInfo StartInfo {get;}
StartTime	Property	datetime StartTime {get;}
SynchronizingObject	Property	System.ComponentModel.ISynchronizeInvoke SynchronizingObject {get;}
Threads	Property	System.Diagnostics.ProcessThreadCollection Threads {get;}
TotalProcessorTime	Property	timespan TotalProcessorTime {get;}
UserProcessorTime	Property	timespan UserProcessorTime {get;}
VirtualMemorySize	Property	int VirtualMemorySize {get;}
VirtualMemorySize64	Property	long VirtualMemorySize64 {get;}
WorkingSet	Property	int WorkingSet {get;}
WorkingSet64	Property	long WorkingSet64 {get;}
Company	ScriptProperty	System.Object Company {get=\$this.MainModule.FolderName}
CPU	ScriptProperty	System.Object CPU {get=\$this.TotalProcessorTime}
Description	ScriptProperty	System.Object Description {get=\$this.MainModule.Name}
FileVersion	ScriptProperty	System.Object FileVersion {get=\$this.MainModule.FileVersion}
Path	ScriptProperty	System.Object Path {get=\$this.MainModule.Path}

With the information, we now create a pipeline that selects objects that must meet two conditions for the object to be selected: 1) the processname must begin with "c" and 2) the cpu must be greater than 10 seconds. This requires using a complex conditional expression in the where-object cmdlet. We construct the pipeline as follows:

```
get-process | where-object { $_.processname -like 'c*' -and $_.cpu -gt 10 }
```

```
gt 10 }
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
4193	93	196692	220784	652	2,245.64	6472	chrome
552	53	97252	91252	410	573.23	6600	chrome
190	30	108340	101264	328	56.41	6656	chrome
179	21	49088	35196	236	16.16	6688	chrome
180	39	204408	196088	397	901.17	6972	chrome
277	46	389004	428892	695	193.67	7856	chrome
237	33	121596	134716	420	57.64	8048	chrome
233	32	133484	146484	324	536.75	8144	chrome
577	67	264576	276904	471	2,377.31	8816	chrome
210	34	140472	172384	405	18.56	9664	chrome
1297	267	1013952	985916	1751	1,589.86	12412	chrome
209	29	92876	101108	306	53.59	14080	chrome
268	41	167052	213392	579	293.02	15348	chrome

The -like operator allows pattern matching on string properties using wildcards. The pattern 'c*' means any string that begins with 'c' and is followed by any number of characters. In the conditional expression above "\$_.ProcessName -like 'c*' " selects process whose name begins with "c" or "C". But for those processes selected we want the ones whose cpu time is greater than 10 seconds. We use the -and operator to join the second conditional expression "\$_.CPU -gt 10".

The above example uses -and to join two conditional expressions. If we need to add more conditions, we may use -and as necessary to add the required conditions, but note, all conditions must be satisfied for the object to be selected and continue through the pipeline.

Using -or:

Let's now look at the case where an object needs to meet one of a set of conditions. The -or operator allows joining multiple conditional expressions and selects the object if one of the expressions evaluates to true. The table below illustrates the result of the -or operator when applied against two values of expressions that evaluate to true or false.

OR	Value of condition _a : False	Value of condition _n : True
Value of condition ₁ : False	Result of cond _a OR Result of cond ₁ : False	Result of cond _n OR Result of cond ₁ : True
Value of condition _n : True	Result of cond _a OR Result of cond _n : True	Result of cond _n OR Result of cond _n : True

As is shown below when joining expressions with -or only one condition has to be true for the complex conditional expression to be true.

```
PS C:\Users>
PS C:\Users> $true -or $true
True
PS C:\Users> $false -or $true -or $false -or $false
True
```

In this example, we want to select all process whose process name begins with "a", "d", or "s". We construct the pipeline below.

```
get-process |
  where-object { $_.processname -like 'a*' -or
                 $_.processname -like 'd*' -or
                 $_.processname -like 's*' }
```

```

PS C:\Users>
PS C:\Users> get-process |
>>   where-object { $_.processname -like 'a*' -or
>>                 $_.processname -like 'd*' -or
>>                 $_.processname -like 's*' }
>>

```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
81	7	1040	1076	44		1772	armsvc
168	10	7276	7544	39	453.88	12076	audiodg
442	19	6488	10500	82		1956	dasHost
115	7	1824	2356	54		4280	dllhost
653	59	139628	60880	574		680	dwm
271	21	5036	8624	112	3.38	7144	ScanToPCActivationA
151	10	2056	6836	43		15048	SearchFilterHost
923	74	47176	49988	698		4200	SearchIndexer
246	10	2296	6444	62	0.02	3612	SearchProtocolHost
329	13	2524	9376	70		15164	SearchProtocolHost
288	10	4276	5280	25		856	services
587	37	17260	15272	680	51.06	5604	SettingSyncHost
666	32	20808	18448	172	6.88	5192	SkyDrive
874	58	96100	92792	357	363.03	6852	Skype
260	10	2172	4716	56		1788	SkypeC2CAutoUpdateS
108	23	37892	30448	84		1812	SkypeC2CPNRSvc
44	2	276	324	4		556	smss
205	14	3384	13548	134	7.19	4560	SnippingTool
1434	120	259204	289980	827	173.19	2188	Solitaire
180	15	4268	10416	133	0.70	13844	splwow64
491	33	5624	9408	86		1592	spoolsv
886	57	40312	48716	272	66.20	6380	Steam
160	11	6412	6692	56		8512	SteamService
337	24	7072	7924	164	1.20	7644	steamwebhelper
557	18	9380	13008	54		980	svchost
740	18	10852	13716	77		1008	svchost
893	31	24636	26140	107		1072	svchost
816	35	22544	25212	118		1100	svchost
2789	77	49344	54820	236		1132	svchost
963	52	21784	26424	153		1204	svchost
672	154	16096	18380	1422		1468	svchost
636	40	30488	31660	167		1648	svchost
354	16	7656	11308	93		1916	svchost
102	8	1472	2004	22		2208	svchost
260	20	6084	9140	91		2268	svchost
549	29	8528	13024	69		3248	svchost
420	34	6388	8876	825		3916	svchost
1296	0	120	2416	9		4	System

What if I wanted processes that met the above conditions but must also meet the condition that cpu time is greater than 50. We construct the pipeline below. We use the parentheses around the or conditions to force PowerShell to evaluate those conditions as a unit. Any objects selected by the complex conditional in the parentheses must also meet the cpu time condition.

```
get-process |
  where-object { ($_.processname -like 'a*' -or $_.processname -like
'd*' -or
    $_.processname -like 's*') -and $_.cpu -gt 50 }
```

```
PS C:\Users>
PS C:\Users> get-process |
>>   where-object { ($_.processname -like 'a*' -or $_.processname -like 'd*' -
>>           $_.processname -like 's*') -and $_.cpu -gt 50 }
>>

Handles  NPM(K)      PM(K)      WS(K)  VM(M)      CPU(s)      Id  ProcessName
----  -----  -----  -----  -----  -----  --
  168      10       7276      7544     39    457.64  12076 audiodg
  568      26      14252     14760    163    51.20   5604 SettingSyncHost
  874      58      96020     92936    356   364.58   6852 Skype
 1434     120     259204    289980    827   173.19  2188 Solitaire
  896      58      40408     49408    274    66.31   6380 Steam
```

To demonstrate the power of regular expressions, we rewrite the above pipeline to use the -match operator and a regular expression that reduces the three conditional expressions joined by -or into one expression.

```
get-process |
  where-object { $_.processname -match '^a|d|s.*' -and $_.cpu -gt
50 }
```

```
PS C:\Users>
PS C:\Users> get-process |
>>   where-object { $_.processname -match '^a|d|s.*' -and $_.cpu -gt 50 }
>>

Handles  NPM(K)      PM(K)      WS(K)  VM(M)      CPU(s)      Id  ProcessName
----  -----  -----  -----  -----  -----  --
  168      10       7276      7544     39    458.09  12076 audiodg
  589      37      16480     16644    681    51.27   5604 SettingSyncHost
  874      58      96020     92936    356   364.89   6852 Skype
 1434     120     259204    289980    827   173.19  2188 Solitaire
  896      58      40404     49404    274    66.31   6380 Steam
```

In the regular expression above, the caret "^" *anchors* the match at the beginning of the string. This means that the next match character must be at the beginning of the matching string. The next match pattern "[a|d|s]" means either "a", "d",

or "s"; the match pattern "^a|d|s" means an "a", "d", "s" at the beginning of the process name. The next match pattern ".*" is equivalent to the wildcard "*" which means match zero or more characters. In fact because of the way regular expressions work, this pattern is unnecessary.

Using -not:

Finally, let's look at the -not operator. The -not operator reverses the true or false value returned by an expression.

```
PS C:\Users>
PS C:\Users> -not $true
False
PS C:\Users> -not $false
True
```

This pipeline returns all file system objects in the specified directory that are not read-only.

```
get-childitem c:\windows\fonts | where-object { -not $_.IsReadOnly }
get-childitem c:\windows\fonts | where-object { ! $_.IsReadOnly }
```

The next example returns all processes whose name does not begin with "c" and whose cpu time is not greater than 10. The parentheses are used to control how PowerShell evaluates the conditions. The inner conditional expression "\$_.processname -like 'c*' -and \$_.cpu -gt 10" is evaluated first. For each object flowing through this stage of the pipeline, this conditional expression evaluates to true or false based upon contents of the processname and cpu property. Once the inner expression is evaluated. The -not operator flips the result; so, if the processname does begin with 'c' and the cpu time is greater than 10, the -not operator flips it to false causing where-object to drop the object.

```
get-process | where-object { -not ($_.processname -like 'c*' -and
$_.cpu -gt 10) }
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
501	47	124968	143668	373	3.95	14168	AcroRd32
542	21	9496	19664	112	0.50	15300	AcroRd32
81	7	1040	1076	44		1772	armsvc
161	10	6972	7528	39	495.14	12076	audiogd
233	17	14924	6984	83		2920	CaptureGenUSB
186	24	64736	77992	282	2.86	4088	chrome
187	24	71880	77168	270	2.69	4840	chrome
176	20	35004	39252	243	2.00	6548	chrome
177	19	28724	15104	222	0.97	6664	chrome
181	20	44004	36004	230	7.70	6672	chrome
170	18	25488	11652	208	0.89	6680	chrome
173	18	25580	12240	209	0.67	6700	chrome
169	19	26840	12812	219	1.06	6708	chrome
176	20	43212	29140	241	5.13	6720	chrome
172	19	28708	15448	222	0.94	6728	chrome
168	18	25596	12156	208	0.80	6760	chrome
170	19	36384	20900	222	2.38	6776	chrome
174	19	37284	24136	228	2.59	6944	chrome
170	18	30532	18188	214	2.02	6980	chrome
170	18	28292	14744	215	0.91	6988	chrome
191	24	62344	73912	260	2.16	11468	chrome
193	26	97108	104768	292	3.78	11472	chrome
192	22	44636	52184	249	1.06	13228	chrome
52	6	852	884	48		2836	conhost
42	5	724	636	24		3028	conhost
60	7	1960	5992	53	0.48	13736	conhost
272	14	2024	2168	47		740	csrss
450	34	2700	11112	275		812	csrss
136	10	2196	2560	51		1880	CtHdaSvc
442	19	6500	10644	82		1956	dasHost
115	7	1824	2356	54		4280	dllhost
701	64	123188	46276	590		680	dwm
742	48	55608	86460	487	29.41	9892	EXCEL
2770	127	90904	131216	790	250.95	4612	explorer
1072	92	195648	197844	1109	14.69	1248	Facebook
922	33	12896	14128	170	4.27	6632	g2mcomm
471	25	9048	7572	145	1.06	7104	g2mlauncher
292	18	4288	3324	123	0.13	6524	g2mstart
197	14	3552	5548	83		1964	GFEExperienceService
236	19	5448	5396	154		2008	HauppaugeTVServer
916	17	4504	8004	103	10.63	1732	HPNetworkCommunicatorCom
246	17	12904	4052	155		1328	HPSupportSolutionsFrameworkService
66	6	876	1228	61	0.02	7676	hpwschd2
0	0	0	4	0		0	Idle
443	20	5508	3424	124	34.63	4540	ipoint
110	8	1604	1764	38		1936	IPROSetMonitor
425	23	8668	2392	151	20.75	4552	itype
1374	40	75612	90552	1339	4,623.75	4844	livecomm
1668	23	12448	18244	60		864	lsass

One last example to demonstrate the use of this operator. Generally we want to avoid expressions of this type since it's very difficult to understand. Note also the use of the nested parentheses. We use this syntax to evaluate the complete conditional expressions in the sequence desired. The innermost parentheses is evaluated first. So, if the process name begins "a", "d", or "s" and the cpu time is greater than 50, the object is discarded otherwise the object is selected.

```
get-process |
    where-object { -not ( ($_.processname -like 'a*' -or
                            $_.processname -like 'd*' -or
                            $_.processname -like 's*') -and
                            $_.cpu -gt 50 ) }
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	ID	ProcessName
501	47	124936	143668	356	3.95	14168	AcroRd32
541	20	9428	19644	110	0.50	15300	AcroRd32
81	7	1040	1076	44		1772	armsvc
233	17	14924	6984	83		2920	CaptureGenUSB
193	29	114168	122056	320	12.73	2088	chrome
200	27	84776	93976	291	11.17	2656	chrome
186	24	64736	77992	282	2.86	4088	chrome
187	24	71880	77180	270	2.69	4840	chrome
4166	89	198632	222508	648	2,434.39	6472	chrome
176	20	35004	39252	243	2.00	6548	chrome
556	52	98460	91616	410	573.45	6600	chrome
190	29	122744	109352	329	58.98	6656	chrome
177	19	28724	15104	222	0.97	6664	chrome
181	20	44004	36004	230	7.70	6672	chrome
170	18	25488	11652	208	0.89	6680	chrome
179	20	40380	32088	233	17.08	6688	chrome
173	18	25580	12240	209	0.67	6700	chrome
169	19	26840	12812	219	1.06	6708	chrome
176	20	43212	29144	241	5.14	6720	chrome
172	19	28708	15448	222	0.94	6728	chrome
168	18	25596	12156	208	0.80	6760	chrome
170	19	36384	20900	222	2.38	6776	chrome
174	19	37284	24136	228	2.59	6944	chrome
180	40	225008	216352	407	1,011.08	6972	chrome
170	18	30532	18188	214	2.02	6980	chrome
170	18	28292	14744	215	0.91	6988	chrome
278	47	310340	345596	768	633.16	7856	chrome
238	33	121196	134080	420	84.75	8048	chrome
233	34	142288	157792	335	732.00	8144	chrome
577	67	269064	280824	473	3,371.11	8816	chrome
							.

For the sake of completeness let's look at -nomatch using the example we previously. The previous example resulted in all process objects whose process name begins with "a", "d", or "s" and have a cpu time greater than 50. By

negating this complex expression, we retrieve the complement set of objects, i.e., the objects that do not meet the condition in the parentheses.

```
get-process |  
  where-object { -not ($_.processname -match '^a|d|s'.*) -and  
  $_.cpu -gt 50) }
```

```
PS C:\Users>  
PS C:\Users> get-process |  
>>   where-object { -not ($_.processname -match '^a|d|s'.*) -and $_.cpu -gt 50 }  
>>  


| Handles | NPM(K) | PM(K)  | WS(K)  | VM(M) | CPU(s)   | ID   | ProcessName   |
|---------|--------|--------|--------|-------|----------|------|---------------|
| 81      | 7      | 1040   | 1076   | 44    |          | 1772 | armsvc        |
| 233     | 17     | 14924  | 7124   | 83    |          | 2920 | CaptureGenUSB |
| 178     | 19     | 28244  | 33136  | 230   | 0.42     | 1888 | chrome        |
| 237     | 37     | 167468 | 181348 | 454   | 19.11    | 2520 | chrome        |
| 280     | 77     | 94988  | 95220  | 873   | 409.41   | 2672 | chrome        |
| 231     | 30     | 113888 | 122612 | 400   | 31.28    | 3648 | chrome        |
| 3977    | 87     | 218256 | 242800 | 617   | 3,028.22 | 6472 | chrome        |
| 176     | 20     | 37812  | 43768  | 244   | 4.02     | 6548 | chrome        |
| 518     | 46     | 66140  | 77852  | 374   | 614.50   | 6600 | chrome        |
| 191     | 29     | 121816 | 106548 | 327   | 76.02    | 6656 | chrome        |
| 177     | 19     | 28724  | 15148  | 222   | 1.03     | 6664 | chrome        |
| 181     | 20     | 36468  | 29604  | 228   | 9.47     | 6672 | chrome        |
| 170     | 18     | 25488  | 11692  | 208   | 1.00     | 6680 | chrome        |
| 179     | 20     | 40004  | 31452  | 233   | 21.14    | 6688 | chrome        |
| 173     | 18     | 25580  | 12280  | 209   | 0.73     | 6700 | chrome        |
| 169     | 10     | 26848  | 12852  | 210   | 1.16     | 6700 | chrome        |


```

Selecting on the absence of a value:

Many times properties of an object do not have a value. To select based upon the absence of a value, we need to compare against the built-in variable \$null. This variable represents the absence of a data. The example below illustrates this concept.

FileInfo objects have a property call *linktype*. For normal files linktype has no value, i.e., is null. For symbolic or hard links, the is property has the type of link. In the image below, note that the file hostslink has a mode of -a---l where the l indicates that the file is a link.

```

PS D:\dir>
PS D:\dir> gci

      Directory: D:\dir

Mode                LastWriteTime         Length  Name
----                -----          -----  --
-a---        5/28/2017   8:29 AM       10111  grid.htm
-a---        6/12/2017   1:41 PM        9729  hosts.xlsx
-a---l       9/17/2017   8:13 PM           0  hostslink

```

In this script, we select all files which are not links

```

get-childitem |
  where-object { $_.LinkType -eq $null }

```

```

PS D:\dir>
PS D:\dir> get-childitem |
>>   where-object { $_.LinkType -eq $null }

      Directory: D:\dir

Mode                LastWriteTime         Length  Name
----                -----          -----  --
-a---        5/28/2017   8:29 AM       10111  grid.htm
-a---        6/12/2017   1:41 PM        9729  hosts.xlsx

```

The next script shows all links.

```

get-childitem |
  where-object { $_.LinkType -ne $null }

```

```

PS D:\dir>
PS D:\dir> get-childitem |
>>   where-object { $_.LinkType -ne $null }

    Directory: D:\dir

Mode                LastWriteTime         Length Name
----                -----          ----- 
-a--- 1      9/17/2017 8:13 PM           0 hostsLink

```

Foreach-Object

Wednesday, September 4, 2013
12:16 PM

<http://technet.microsoft.com/en-us/library/ee176828.aspx>

The foreach-object cmdlet (%) is used somewhere after the beginning of the pipeline and acts upon the individual objects flowing through the pipeline.

In this first example, we pipe an array of consisting of three strings into foreach-object. Within the script block, the string "The color is " is prepended to the current object in the pipeline, creating a new string object is then emitted by foreach-object to be implicitly displayed on the console by out-default.

```
'red', 'green', 'blue' | foreach-object { "The color is " + $_ }
```

```

PS C:\Users>
PS C:\Users> 'red', 'green', 'blue' | foreach-object { "The color is " + $_ }
The color is red
The color is green
The color is blue
PS C:\Users>
```

The following example illustrates how a cmdlet may be executed multiple times using foreach-object with different parameter values. In this case, we execute get-childitem twice. In the first execution -path is supplied a value of "c:\windows\system". In the second, the -path value is "c:\windows\system32".

```
'c:\windows\system', 'c:\windows\system32' |  
    foreach-object { get-childitem -path $_ -filter *.exe }
```

```
PS C:\Users>  
PS C:\Users> 'c:\windows\system', 'c:\windows\system32' |  
    foreach-object { get-childitem -path $_ -filter *.exe }  
  
    Directory: C:\windows\system32  
  
Mode                LastWriteTime       Length Name  
----                -----          ----  --  
-a---]      7/10/2015  6:59 AM        23040 acu.exe  
-a---]      8/18/2015  3:04 AM      1234944 aitstatic.exe  
-a---]      7/10/2015  6:59 AM        97792 alg.exe  
-a---]      7/10/2015  7:00 AM       19456 appidcertstorecheck.exe  
-a---]      7/10/2015  7:00 AM      161280 appidpolicyconverter.exe  
-a---]      7/10/2015  7:00 AM       43416 ApplicationFrameHost.exe  
-a---]      7/10/2015  6:59 AM       26112 ARP.EXE  
-a---]      7/10/2015  7:00 AM       29696 at.exe  
-a---]      7/10/2015  7:00 AM       54272 AtBroker.exe  
-a---]      7/10/2015  6:59 AM       20992 attrib.exe  
-a---]      7/10/2015  7:00 AM      372512 audiogd.exe  
-a---]      7/10/2015  7:00 AM       66560 auditpol.exe  
-a---]      7/10/2015  7:00 AM      140536 AuthHost.exe  
-a---]      7/10/2015  6:59 AM       944640 autochk.exe  
-a---]      7/10/2015  6:59 AM      931840 autoconv.exe  
-a---]      7/10/2015  6:59 AM      899072 autofmt.exe  
-a---]      7/10/2015  6:59 AM       45568 AutoWorkplace.exe  
-a---]      7/10/2015  7:00 AM       60416 AxInstUI.exe  
-a---]      7/10/2015  9:28 AM      112640 baaupdate.exe  
-a---]      7/10/2015  7:00 AM       19808 backgroundTaskHost.exe  
-a---]      7/10/2015  6:59 AM       36864 BackgroundTransferHost.exe  
-a---]      8/15/2015  2:25 AM      242528 bootdvr.exe
```

Let's look at the example below where we select only exe files in the designated directory. We use foreach-object to process each fileinfo object and display the name property and the calculated item shown below. In the foreach scriptblock, the format operator -f is used to display the name property and the derived length property in the console.

```
Get-ChildItem c:\windows -file |
```

```
where-object { $_.Extension -eq '.exe'} |  
    foreach-object { "{0,-20} {1,10}" -f $_.Name,  
    ($_.Length/1MB) }
```

The Foreach-Object cmdlet is also used so frequently that the "%" alias was created. The example above may be written using the "%" alias for foreach-object and the "?" alias for the where-object.

```
Get-ChildItem c:\windows -file |  
    ? { $_.Extension -eq '.exe'} |  
    % { "{0,-20} {1,10}" -f $_.Name, ($_.Length/1MB) }
```

```
PS C:\Users>  
PS C:\Users> Get-ChildItem c:\windows -file |  
>>>    ? { $_.Extension -eq '.exe'} |  
>>>    % { "{0,-20} {1,10}" -f $_.Name, ($_.Length/1MB) }  
bfsvc.exe          0.06787109375  
dchcfg32.exe      0.339210510253906  
dchcfg64.exe      0.406593322753906  
dcmdev64.exe      0.218116760253906  
DfsrAdmin.exe     0.24609375  
explorer.exe      3.080078125  
fveupdate.exe     0.0146484375  
hapint.exe        0.466163635253906  
hapint64.exe      0.466163635253906  
HelpPane.exe      0.69970703125  
hh.exe            0.01611328125  
notepad.exe       0.1845703125  
regedit.exe       0.4072265625  
splwow64.exe      0.06396484375  
twunk_16.exe      0.0473785400390625  
twunk_32.exe      0.02978515625  
winhlp32.exe      0.00927734375  
write.exe         0.009765625  
PS C:\Users>
```

In the next example below we use the range operator in the first pipeline to generate an array of integers with containing the values 1 through 5. The second pipeline has incorrect syntax because PowerShell doesn't know how to act upon the integer objects flowing through the pipeline. The third pipeline shows the proper use of foreach-object to act upon the individual objects. The last example produces the same result as the previous pipeline but in this case the alias % is used instead of foreach-object,

```
1..5                                # create integer array with  
values 1 through 10  
# Supress errors because the following statement causes errors
```

```
$erroractionpreference = 'SilentlyContinue'
1..5 | write-host "The number is " $_ # this
doesn't work
$erroractionpreference = 'Continue'
1..5 | foreach-object { write-host "The number is" $_ } # this
does
1..5 | % { write-host "The number is $_ " } # using
the % alias
```

```
PS C:\Users>
PS C:\Users> 1..5 # create integer array with values 1
1
2
3
4
5
PS C:\Users> # Supress errors because the following statement causes errors
PS C:\Users> $erroractionpreference = 'SilentlyContinue'
PS C:\Users> 1..5 | write-host "The number is " $_ # this doesn't
PS C:\Users> $erroractionpreference = 'Continue'
PS C:\Users> 1..5 | foreach-object { write-host "The number is" $_ } # this does
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
PS C:\Users> 1..5 | % { write-host "The number is $_ " } # using the %
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
```

In this example, we pipe the integer array 1,3,5,7,9 into foreach-object. The script block increments the current object (integer) and displays it to the console.

```
1,3,5,7,9 | foreach-object { write-host "$($_ + 1)" }
```

```
PS C:\Users>
PS C:\Users> 1,3,5,7,9 | foreach-object { write-host "$($_ + 1)" }
2
4
6
8
10
```

For someone new to PowerShell the inclination is to use the syntax *write-host "\$_ + 1"* in the above pipeline. However, as can be seen below, this does not produce the desired result since write-host would evaluate \$_ +, and 1 independently of

each other. We need to enclose the calculation in the subexpression `$(_ + 1)`. This forces PowerShell to evaluate the calculation subexpression.

```
PS C:\Users>
PS C:\Users> 1,3,5,7,9 | foreach-object { write-host "$_ + 1" }
1 + 1
3 + 1
5 + 1
7 + 1
9 + 1
```

But since PowerShell displays the value of a variable when the variable name stands alone, the above working pipeline may be rewritten as follows:

```
1,3,5,7,9 | foreach-object { $_ + 1 }
```

```
PS C:\Users>
PS C:\Users> 1,3,5,7,9 | foreach-object { $_ + 1 }
2
4
6
8
10
```

We may include multiple lines of scripting code in a script block. In this example, we total all the integers following through the pipeline in the variable \$n. Through each iteration of the of foreach, we display the current integer object. After the pipeline completes the total is displayed.

```
51..61 | foreach-object {
    write-host "The current value is $_"
    $n = $n + $_ ;
}
write-host "The total is $n"
```

```
PS C:\Users>
PS C:\Users> 51..61 | foreach-object {
>>>         write-host "The current value is $_"
>>>         $n = $n + $_ ;
>>>     }
The current value is 51
The current value is 52
The current value is 53
The current value is 54
The current value is 55
The current value is 56
The current value is 57
The current value is 58
The current value is 59
The current value is 60
The current value is 61
PS C:\Users>
PS C:\Users> write-host "The total is $n"
The total is 1848
PS C:\Users>
```

When a scriptblock contains multiples statements, a semicolon is required after each statement whenever statements appear on the same line. In the above example, the script code is on separate lines so the semicolon is not required. To complete the above example, we change the script block to also count the number of integers that flow through the pipeline. The script below also demonstrates a best practice in that the variables are descriptive of their function. In addition it shows the algorithms necessary to count objects and to total objects. One very important point about the script below, note that the variables \$total and \$count are initialized to 0 at the beginning of the script.

```
$total = $count = 0
51..61 | foreach-object {
    $total = $total + $_
    $count = $count + 1
    write-host "The current value is $_"
}
write-host "The number of integers processed is $count`nThe total of
all integers is $total"
```

```
PS C:\Users>
PS C:\Users> $total = $count = 0
PS C:\Users> 51..61 | foreach-object {
>>>         $total = $total + $_
>>>         $count = $count + 1
>>>         write-host "The current value is $_"
>>>     }
The current value is 51
The current value is 52
The current value is 53
The current value is 54
The current value is 55
The current value is 56
The current value is 57
The current value is 58
The current value is 59
The current value is 60
The current value is 61
PS C:\Users> write-host "The number of integers processed is $count`nThe total of all integers is $total"
The number of integers processed is 11
The total of all integers is 616
PS C:\Users>
```

The above script initialized the variables \$total and \$count at the beginning of the script this makes the script repeatable.

In the example below note the effect of not initializing the variable and attempting to repeat a pipeline. We would expect the second execution of the pipeline would produce the same result. However, since \$n was not initialized to 0 before the second execution, it contains the total from the first execution resulting in a non-repeatable outcome.

```
1..5 | foreach-object { $n = $n + $_ }
write-host "The total is $n"
1..5 | foreach-object { $n = $n + $_ }
write-host "The total is $n"
```

```
PS C:\Users>
PS C:\Users> 1..5 | foreach-object { $n = $n + $_ }
PS C:\Users>
PS C:\Users> write-host "The total is $n"
The total is 15
PS C:\Users> 1..5 | foreach-object { $n = $n + $_ }
PS C:\Users> write-host "The total is $n"
The total is 30
```

The next script shows another example of the use of foreach-object, in this script we concatenate the multiple strings in the array \$words into one string.

```

# $words is a string array (or collection) that has all the assigned words
$words =
'The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog'
# each word in $words is piped to foreach where we concatenate (join) each word
$words | % { $sentence = $sentence + $_ + ' ' }
# display the sentence
write-host "The joined string is '$sentence'" -foreground magenta

```

```

PS C:\Users>
PS C:\Users> # $words is a string array (or collection) that has all the assigned words
PS C:\Users> $words = 'The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog'
PS C:\Users> # each word in $words is piped to foreach where we concatenate (join) each word
PS C:\Users> $words | % { $sentence = $sentence + $_ + ' ' }
PS C:\Users> # display the sentence
PS C:\Users> write-host "The joined string is '$sentence'" -foreground magenta
The joined string is 'The quick brown fox jumps over the lazy dog'

```

In example below, the pipeline shows the service name and the boolean service property canstop. In the script block, the metacharacter `t is used in the display to the console to display of the canstop property at the next tab stop following the display of the name property.

```

get-service | select-object -first 5 |
    foreach-object { "{0,-35} {1,-8}" -f $_.name, $_.canstop }

```

```

PS C:\Users>
PS C:\Users> get-service | select-object -first 5 |
    >>>     foreach-object { "{0,-35} {1,-8}" -f $_.name, $_.canstop }
AdobeARMservice           True
AdobeFlashPlayerUpdateSvc False
AeLookupSvc                True
ALG                         False
AMD External Events Utility True
PS C:\Users>

```

This pipeline is a variation of the above example but combines the use of where-object and foreach-object to filter the service objects by selecting running services and display the name and the property canstop. We also use the aliases for the where-object and select-object cmdlet.

```
get-service | ? { $_.status -eq 'Running'} | % { "{0,-35} {1,-8}" -f  
$_.name, $_.canstop }
```

```
PS C:\Users>  
PS C:\Users> get-service | ? { $_.status -eq 'Running'} | % { "{0,-35} {1,-8}" -f $_.name, $_.canstop }  
AdobeARMservice True  
AeLookupSvc True  
AMD External Events Utility True  
Appinfo True  
AudioEndpointBuilder True  
AudioSrv True  
BFE True  
BITS True  
Browser True  
CertPropSvc True  
ClickToRunSvc True  
CryptSvc True  
CscService True  
DcomLaunch False  
Dhcp True  
DiagTrack True  
Dnscache True  
DPS True  
EapHost True
```

The script below demonstrates the use of multiple lines and indentation to make the script more readable. As the complexity and thus the number of lines in a script grows, using spacing, indentation, and comments is important for readability. This example also use the semicolon ";" to separate the write-host statements on the same line.

```
get-childitem -file -recurse |  
foreach-object {  
    write-host $_.directoryname;    write-host `t `t $_.basename  
}
```

```
PS C:\dell>
PS C:\dell> get-childitem -file -recurse |  
->     foreach-object {  
->       write-host $_.directoryname;   write-host `t `t $_.basename  
-> }  
->  
C:\dell\drivers\R227150  
    megaraid  
C:\dell\drivers\R227150  
    Mraid35x  
C:\dell\drivers\R227150  
    NODEV  
C:\dell\drivers\R227150  
    oemsetup  
C:\dell\drivers\R227150  
    R227150  
C:\dell\drivers\R227150  
    Version  
C:\dell\Printers\EMBXPS\drivers\print\xps  
    DKACII50  
C:\dell\Printers\EMBXPS\drivers\print\xps  
    DKACII50  
C:\dell\Printers\EMBXPS\drivers\print\xps
```

However, it's generally not a good idea to have multiple statements on a single line since this would make the script more difficult to read. So, a better form of the script is shown below where the second write-host statement is on a separate line eliminating the need for the semicolon.

```
get-childitem -file -recurse |  
foreach-object {  
    write-host $_.directoryname  
    write-host `t `t $_.basename  
}
```

One issue with this script is that it repeatedly displays the directory name, let's alter it so the directory name is displayed only when it changes. The script below use the IF statement (more on this later) to compare the current directory name with directory name saved when a directory name change. The script code checks the directory name of the current file system object, if the name is the same as the previously saved directory name, then the directory name is not displayed. If it isn't, we display the directory name followed by the name (basename) of the file system object indented on the next line. We also save the current directory name as the previously save directory name. What is shown

below is commonly called a *break*. In this case, we are *breaking* on the directory name.

```
get-childitem -file -recurse |  
    foreach-object {  
        if ($.DirectoryName -eq $previousname) {  
            write-host `t $_.basename      # it's the same so don't  
display the dir name  
        } else {  
            write-host $_.DirectoryName      # it's different so display  
the name  
            $previousname = $_.DirectoryName # need so save the current  
name to compare against  
            write-host `t $_.basename      # display the file name  
        }  
    }  
}
```

```
PS C:\dell>  
PS C:\dell> get-childitem -file -recurse |  
>>    foreach-object {  
>>        if ($.DirectoryName -eq $previousname) {  
>>            write-host `t $_.basename      # it's the same so don't display t  
>>        } else {  
>>            write-host $_.DirectoryName      # it's different so display the na  
>>            $previousname = $_.DirectoryName # need so save the current name to  
>>            write-host `t $_.basename      # display the file name  
>>        }  
>>    }  
>>  
C:\dell\drivers\R227150  
    megaraid  
    Mraid35x  
    NODEV  
    oemsetup  
    R227150  
    Version  
C:\dell\Printers\EMBXPS\drivers\print\XPS  
    DKACII50  
    DKACII50  
    DKACII50  
    DKACII50  
    DKACII50  
    DKACII50P  
    DKACII50W  
    DKACII5A  
    DKACIIF0_PipelineConfig  
    DKACIIP0  
    DKACIIP0C  
    SETUP  
C:\dell\Printers\EMBXPS\drivers\print\XPS\AMD64  
    DKACIIFA
```

Notice the difference between this illustration and the previous one. This script provides a less busier display on the console by showing the directory name only once for each file in the directory.

Let's take a look at example that *parses* a csv file. PowerShell makes this process easy by the import-csv. This cmdlet creates a custom object with the csv column names as properties. Using the sample of a firewall log below.

Date	Time	Syslog_ID	Source_IP	Source_Port	Dest_IP	Dest_Port
Jan 16 2014	14:54:31	106023	188.26.62.236	30870	207.75.134.154	8832
Jan 16 2014	14:54:31	106023	193.110.18.33	62629	207.75.134.154	8832
Jan 16 2014	14:54:30	106023	187.13.238.91	22350	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	46.249.171.169	27376	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	188.134.40.93	49611	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	187.13.238.91	13757	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	80.99.73.186	16008	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	187.13.238.91	48884	207.75.134.154	8832

```

PS C:\Users>
PS C:\Users> import-csv .\firewall_log.csv | get-member

    TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        -----      -----
Equals      Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType     Method      type GetType()
ToString    Method      string ToString()
Date        NoteProperty string Date=Jan 16 2014
Dest_IP    NoteProperty string Dest_IP=207.75.134.154
Dest_Port   NoteProperty string Dest_Port=8832
Source_IP   NoteProperty string Source_IP=188.26.62.236
Source_Port NoteProperty string Source_Port=30870
Syslog_ID  NoteProperty string Syslog_ID=106023
Time        NoteProperty string Time=14:54:31

```

We now may use the pipeline below to display the pertinent fields from the log.

```

import-csv firewall_log.csv |
  foreach-object {
    $_.Source_ip+':'+$_.Source_port+"`t=>`t"+$_.Dest_IP+':'+$_.Dest_port
  }

```

```

PS C:\Users>
PS C:\Users> import-csv firewall_log.csv |
  >>> foreach-object { $_.Source_ip+':'+$_.Source_port+"`t=>`t"+$_.Dest_IP+':'+$_.Dest_Port
188.26.62.236:30870      =>      207.75.134.154:8832
193.110.18.33:62629      =>      207.75.134.154:8832
187.13.238.91:22350      =>      207.75.134.154:8832
46.249.171.169:27376      =>      207.75.134.154:8832
188.134.40.93:49611      =>      207.75.134.154:8832
187.13.238.91:13757      =>      207.75.134.154:8832
80.99.73.186:16008       =>      207.75.134.154:8832
187.13.238.91:48884      =>      207.75.134.154:8832

```

The next example illustrates the use of a hash table to easily enumerate the file extensions in a directory. The cmdlet get-childitem retrieves all the file system objects in the directory. The objects are piped into foreach-object. In that script block, we index into the hash table using the extension of the current object and add one to the current entry; the normal algorithm for counting. However, this script as it stands throws an error because the variable \$exts doesn't exist and consequently the type is unknown. PowerShell attempts to index into \$exts but doesn't know whether to add an entry or update an existing entry.

```
gci c:\windows\system32 |  
    foreach-object { $exts[$_.extension] = $exts[$_.extension] + 1 }  
$exts
```

```
PS C:\Users>  
PS C:\Users> gci c:\windows\system32 |  
->>>     foreach-object { $exts[$_.extension] = $exts[$_.extension] + 1 }  
Cannot index into a null array.  
At line:2 char:21  
+     foreach-object { $exts[$_.extension] = $exts[$_.extension] + 1 }  
+  
+     + CategoryInfo          : InvalidOperation: (:) [], RuntimeException  
+     + FullyQualifiedErrorId : NullArray  
  
Cannot index into a null array.  
At line:2 char:21  
+     foreach-object { $exts[$_.extension] = $exts[$_.extension] + 1 }  
+  
+     + CategoryInfo          : InvalidOperation: (:) [], RuntimeException  
+     + FullyQualifiedErrorId : NullArray
```

We correct this script by initializing the variable \$exts as a *null* hash table, i.e., empty hash table. This is an important and required step. When the first object is processed, the script block indexes into \$exts successful since PowerShell knows that, at this point, \$exts is an empty has table causing PowerShell to add the first entry to the hash table.

```
$exts = @{}  
gci c:\windows\system32 |  
    foreach-object { $exts[$_.extension] = $exts[$_.extension] + 1 }  
$exts
```

```

PS C:\Users>
PS C:\Users> $exts = @{}
PS C:\Users> gci c:\windows\system32 |
>>>   foreach-object { $exts[$_.extension] = $exts[$_.extension] + 1 }
PS C:\Users> $exts

Name          Value
----          ---
.rs           13
.exe          547
.ime          1
.sys          3
.vbs          3
.tlb          10
.sdi          1
.txt          1
.com          5
.sdb          1
.mof          11
.iec          1
.NLS          120
.efi          4
.xsl          3
.msc          21
.h            1
.rtl          2
.din          1
.rat          2
.Shared       2
.acm          6

```

This example is just a rewrite of the above script to use the post increment operator to count the extensions.

```

$exts = @{}
gci c:\windows\system32 |
  foreach-object { $exts[$_.extension]++ }
$exts

```

```

PS C:\Users>
PS C:\Users> $exts = @{}
PS C:\Users> gci c:\windows\system32 |
>>>     foreach-object { $exts[$_.extension]++ }
PS C:\Users>
PS C:\Users> $exts

```

Name	Value
.rs	13
.exe	547
.ime	1
.sys	3
.vbs	3
.tlb	10
.sdi	1
.txt	1
.com	5
.sdb	1
.mof	11
.iec	1
.NLS	120
.efi	4
.xsl	3
.msc	21
.h	1
.r11	2
.din	1
.rat	2
.Shared	2
.acm	6

This script finds files with a .docx extension (Word documents) and creates a hash table with the file name as the index and the file size (length) as the value. This script uses the % alias for foreach-object. Note also the use of the -filter switch. This switch causes the cmdlet to pass the wildcard pattern to the PowerShell File System Provider (more on PSProviders later) where the file selection is done. If this switch is not used

```

set-location c:\
$WordFiles = @{}
gci -filter *.docx -recurse -erroraction SilentlyContinue |
    % {$WordFiles[$_.name] = $_.length}
$WordFiles

```

```

PS C:\users>
PS C:\users> $WordFiles = @{}
PS C:\users> gci -filter *.docx -erroraction SilentlyContinue | 
>>> % {$WordFiles[$_.name] = $_.length}
PS C:\users> $WordFiles

Name                           Value
----                           ---
resume.docx                     60
sales_na.docx                   60
sales_sa.docx                   60

```

Let's look at another example of using foreach-object and hash tables. Assume we want to check if a group of servers are connected (kinda of a heartbeat check). The server names and IP address are in a csv file as shown below.

Server	IPAddress
lab-gw	207.75.134.11
WCC-web	198.111.176.6
google	8.8.8.8

We want to load the file in a hash table and ping the associated ip address. To minimize noise, we only want to show the reply line. In the script below, note the use of the GetEnumerator method. This method pipes the individual entries in the hash table down the pipeline. If the GetEnumerator method is not used the complete hash table would be piped as one object. This method is always required when piping a hash table.

```

$servers = @{}
import-csv servers.csv | % { $servers[$_.server] = $_.ipaddress }
$servers.GetEnumerator() | % { ping $_.value -n 1} | select-string
"Reply"

```

```

PS C:\users>
PS C:\users> $servers = @{}
PS C:\users> import-csv servers.csv | % { $servers[$_.server] = $_.ipaddress }
PS C:\users> $servers.GetEnumerator() | % { ping $_.value -n 1} | select-string
"Reply"
Reply from 8.8.8.8: bytes=32 time=74ms TTL=43
Reply from 207.75.134.1: bytes=32 time=81ms TTL=239

```

In our next example, we would like to create a range of characters, every similar to generating a range of integers using the range operator. The range operator (..) generates a range of integers. In the example below, we create a range of integers range from 150 through 160.

```
150..160
```

```
PS C:\users>
PS C:\users> 150..160
150
151
152
153
154
155
156
157
158
159
160
```

It would be nice if we could use the range operator to produce a range of characters as in 'a'..'z'. Ideally, we would like this syntax to produce the range of characters from 'a' through 'z'. Unfortunately, PowerShell doesn't support this functionality.

```
PS C:\users>
PS C:\users> 'a'..'z'
Cannot convert value "a" to type "System.Int32". Error: "Input string was not in a co
At line:1 char:1
+ 'a'..'z'
+ ~~~~~
+ CategoryInfo          : InvalidArgument: (:) [], RuntimeException
+ FullyQualifiedErrorId : InvalidCastFromStringToInteger
```

The error is thrown because the literals 'a' and 'z' are strings and not individual characters. The range operator .. Forces PowerShell to attempt to recast the string 'a' into an integer. But the string 'a' cannot be recast to an integer so the

error is thrown. We can however use a bit of PowerShell trickery to get a series of characters.

We know that the syntax 'a' generates a string containing one character. PowerShell supports the [char] type that represents one Unicode character.

```
PS C:\users>
PS C:\users> [char] $c = 'c'
PS C:\users> $c | gm

TypeName: System.Char

Name      MemberType Definition
----      -----  -----
CompareTo Method   int CompareTo(System.Object value), int CompareTo(char v
Equals    Method   bool Equals(System.Object obj), bool Equals(char obj), b
GetHashCode Method  int GetHashCode()
GetType   Method   type GetType()
GetTypeCode Method  System.TypeCode GetTypeCode(), System.TypeCode IConverti
.ToBoolean Method  bool IConvertible.ToBoolean(System.IFormatProvider provider)
ToByte    Method   byte IConvertible.ToByte(System.IFormatProvider provider)
ToChar    Method   char IConvertible.ToChar(System.IFormatProvider provider)
ToDateTime Method  datetime IConvertible.ToDateTime(System.IFormatProvider provider)
.ToDecimal Method  decimal IConvertible.ToDecimal(System.IFormatProvider provider)
.ToDouble  Method   double IConvertible.ToDouble(System.IFormatProvider provider)
ToInt16   Method   int16 IConvertible.ToInt16(System.IFormatProvider provider)
ToInt32   Method   int IConvertible.ToInt32(System.IFormatProvider provider)
```

We also know that the syntax below types the 'a' as a character as shown by the GetType method. The syntax [int] [char] 'a' causes PowerShell to recast the string 'a' into a character type; then PowerShell attempts to recast the character 'a' to an integer. This is a well-defined in PowerShell (and other programming languages). What PowerShell does in this case is to recast the character 'a' into its ASCII decimal equivalent 97 (see <http://www.rapidtables.com/code/text/ascii-table.htm>).

```
[char]'a'
([char]'a').GetType()
[int] [char] 'a'
```

```
PS C:\users>
PS C:\users> [char]'a'
a
PS C:\users> ([char]'a').Gettype()
IsPublic IsSerial Name                                     BaseType
----- -----   -- System.ValueType
True      True    Char

PS C:\users>
PS C:\users> [int] [char] 'a'
97
```

Based upon what we just learned, it would seem that we should produce a range of decimal ASCII values of the desired range 'a' through 'z' using the syntax below.

```
[int] [char] 'a'..[int] [char] 'z'
```

```
PS C:\users>
PS C:\users> [int] [char] 'a'..[int] [char] 'z'
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
```

Because of the way PowerShell works, we may reduce the syntax to that below because the range operator ".." causes PowerShell to attempt to recast the operands to integers. Since a character may be recast to its integer decimal ASCII equivalent, the desired result is produced.

```
[char] 'a'..[char] 'z'
```

```
PS C:\users>
PS C:\users> [char] 'a'..[char] 'z'
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
```

The converse of what we demonstrated above also works. An integer may be recast into its corresponding ASCII character equivalent provided that the integer has a value of 0 to 255. Note, since PowerShell internally uses UTF-16, of which ASCII is a subset, integers ranging from 0 to 65535 may be recast to characters.

```
[char] 97
```

```
[char] 122
```

```
PS C:\users>
PS C:\users> [char] 97
a
PS C:\users>
PS C:\users> [char] 122
z
PS C:\users>
```

We may now write the pipeline below to produce the range of characters from 'a' too 'z'. The first part of the pipeline generates the integers corresponding to the ASCII values 'a' through 'z'. The second part of the pipeline simply use foreach to recast the integer back to its ASCII character equivalent.

```
[char] 'a'..[char] 'z' | foreach-object { [char] $_ }
```

```
PS C:\users>
PS C:\users> [char] 'a'..[char] 'z' | foreach-object { [char] $_ }
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
```

Enhanced Syntax:

As seen in previous examples, many times before a pipeline is executed, variables need to be initialized before executing the pipeline and executing some related code after the pipeline has executed. This is so common that the `foreach-object` was enhanced with additional syntax. The `foreach-object` cmdlet syntax supports a `begin`, `process`, and `end` script blocks as described below.

```
foreach-object -begin <scriptblock> -process <scriptblock> end -end
<scriptblock>
```

The `-begin` block is executed once before objects are placed in the pipeline. The `-process` block is executed for each object flowing through the pipeline. The `-end` block is executed once after all objects have flowed through the pipeline.

The example below best illustrates this syntax. The pipeline counts the number of files and calculates the total size of the files in the `c:\windows` directory. The variables `$count` and `$total` are initialized to 0 in the `begin` block. This only happens before objects are placed in the pipeline. In the `process` block, the variable `$count` is incremented. The length of the current `FileInfo` object in the pipeline is added to the variable `total`. The `end` block is executed after all `FileInfo` objects are `process` and displays the variable on the console.

```
get-childitem c:\windows |  
foreach-object -begin { $total = $count = 0 } `  
    -process { $total += $_.length  
               $count++ }`  
    -end {write-host "$count files in c:\windows with  
total size of $total" }
```

```
PS C:\Users>  
PS C:\Users> get-childitem c:\windows |  
>> foreach-object -begin { $total = $count = 0 } `  
>>     -process { $total += $_.length  
>>                 $count++ }`  
>>     -end {write-host "$count files in c:\windows with total size of $total"  
110 files in c:\windows with total size of 9655492  
PS C:\Users>
```

Alternate Destinations

Monday, January 28, 2013
10:12 PM

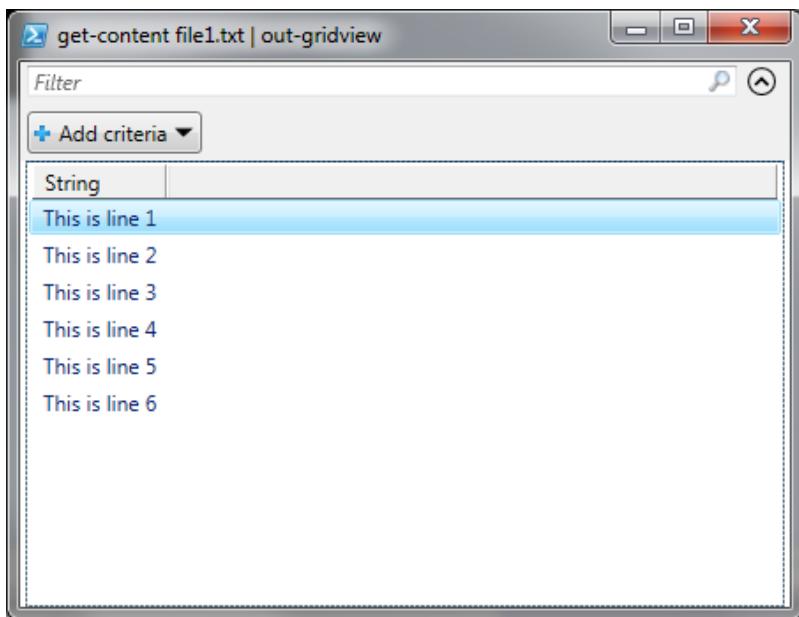
The default destination of objects exiting the pipeline is the console which is defined as *out-default*.

Other destinations are possible through the use of the *out* cmdlets.

out-host	Identical to out-default, the console.
out-file	The destination is the text file specified to the cmdlet. The encoding of the file is typically UTF-16 which is a double-byte Unicode encode. The output is equivalent to a double-byte ASCII file which is compatible with most text editors. If there is a need for a UTF-8 single byte file using the switch -Encoding ASCII.
out-gridview	The destination is a GUI window that allows selection of viewable content.
out-null	As the name implies, the object exiting the pipeline is discarded.
out-printer	Sends output to the default printer or to an alternate printer as indicated by the -Name <printer name> parameter. The <printer name> must be the name of a printer found in "Devices and Printers".
out-string	The objects exiting the pipeline are converted to an array of strings.

```
get-content file1.txt | out-gridview
```

```
PS C:\Users>
PS C:\Users> get-content file1.txt | out-gridview
PS C:\Users> get-content file1.txt | out-gridview
PS C:\Users>
```



The `-stream` parameter of `out-string` alters the behavior so that instead of the input objects comprising one large string, each individual object is emitted as a string. This is best illustrated by the sequence of below.

```

PS C:\Users> get-content file2.txt
A Line 01
B Line 02
C Line 03
D Line 04
E Line 05
F Line 06
PS C:\Users> $f1 = get-content file2.txt
PS C:\Users> $f1.GetType()                                This creates a string array of 6
                                                               elements. Each element holds one
                                                               BaseType
                                                               -----
                                                               System.Array

IsPublic IsSerial Name
----- ----- -----
True     True      Object[]

PS C:\Users> $f1[0]
A Line 01
PS C:\Users>
PS C:\Users> $f2 = get-content file2.txt | out-string
PS C:\Users> $f2.GetType()                                out-string creates one big
                                                               string from each line in the
                                                               BaseType
                                                               -----
                                                               System.Object

IsPublic IsSerial Name
----- ----- -----
True     True      String

PS C:\Users> $f2[0]
A
PS C:\Users>

```

Formatting

Monday, January 28, 2013
5:55 PM

Creating derived properties:

The pipeline below provides a display of all fileInfo objects on the console. The -File switch parameter causes the get-childitem cmdlet to emit only fileInfo objects.

```
get-childitem c:\windows -file
```

The one issue with the output of the above is that the file size (length) is represented as bytes. Since many of the files in this directory are megabytes in size, we would like display the size in megabytes. We use the select-object cmdlet to select the name and to create the *derived* property. The code in the script block divides the file size for the current fileInfo object by 1mb and produces the *synthetic* property of the size in megabytes. The cmdlet then creates a fileInfo object consisting of the name and `$_.length / 1mb` property and passes it down the pipeline.

```
get-childitem c:\windows -file | select-object name, { $_.length /  
1mb }`  
| format-table -autosize
```

```
PS C:\users>  
PS C:\users> get-childitem c:\windows -file | select-object name, { $_.length /  
>> 1mb }`  
| format-table -autosize  
  
Name          $_.length / 1mb  
---  
atiogl.xml      0.0174837112426758  
ativpsrm.bin        0  
bfsvc.exe        0.06787109375  
bootstat.dat      0.064453125  
csup.txt        1.1444091796875E-05  
DfsrAdmin.exe      0.24609375  
DfsrAdmin.exe.config  0.00125408172607422  
DirectX.log        0.0306711196899414  
DPINST.LOG        0.0119705200195313  
DtcInstall.log      0.00193309783935547  
eplauncher.mif      0.00185489654541016  
explorer.exe        2.73876953125  
fveupdate.exe       0.0146484375  
HelpPane.exe        0.69970703125  
hh.exe            0.01611328125  
IE10_main.log       0.00596809387207031  
IE11_main.log       0.00723075866699219  
IE9_main.log        0.00345611572265625  
iis7.log           0.0280580520629883  
MEMORY.DMP         722.334157943726  
mib.bin            0.041132926940918  
msdfmap.ini        0.00133991241455078  
msxml4-KB954430-enu.LOG 0.280910491943359  
msxml4-KB973688-enu.LOG 0.280361175537109  
msxml4-KB973688-enu.LOG 0.280361175537109
```

We pipe the output of select-object into format-table to produce a tabular list to the console. The -autosize switch parameter instructs format-table to eliminate extraneous spaces between the displayed columns. This makes the derived property visible in the console. However, the problem with the above is that when format-table display the derived property the column label is shows the expression and is not very descriptive.

PowerShell supports passing hash tables to certain cmdlets to provide more information to the cmdlet about how to handle parameters. The select-object cmdlet is one of these. We may pass a hash table to select-object to provide information to the cmdlet about the derived property. In the example below, the "Name" key provides the name of the property. The "Expression" provides the arithmetic expression that calculates the value associated with "Name".

```
gci c:\windows -file |  
    select-object name, length, @{Name='MB';Expression={$_ .length /  
        1mb}} |  
    format-table -auto
```

```

PS C:\users>
PS C:\users> gci c:\windows -file |
>>> select-object name, length, @{Name='MB';Expression={$_.length / 1mb}}
>>> format-table -auto

Name                           Length          MB
----                          ----          --
atiogl.xml                     18333        0.0174837112426758
ativpsrm.bin                   0             0
bfsvc.exe                      71168        0.06787109375
bootstat.dat                   67584        0.064453125
csup.txt                        12            1.1444091796875E-05
DfsrAdmin.exe                  258048       0.24609375
DfsrAdmin.exe.config           1315         0.00125408172607422
DirectX.log                     32161        0.0306711196899414
DPINST.LOG                      12552       0.0119705200195313
DtcInstall.log                 2027         0.00193309783935547
eplauncher.mif                 1945         0.00185489654541016
explorer.exe                    2871808      2.73876953125
fveupdate.exe                   15360        0.0146484375
HelpPane.exe                    733696       0.69970703125
hh.exe                          16896        0.01611328125
IE10_main.log                   6258         0.00596809387207031
IE11_main.log                   7582         0.00723075866699219
IE9_main.log                     3624         0.00345611572265625
iis7.log                         29421       0.0280580520629883
MEMORY.DMP                      757422262    722.334157943726
mib.bin                         43131        0.041132926940918
msdfmap.ini                      1405         0.00133991241455078
msxml4-KB954430-enu.LOG          294556      0.280910491943359

```

Another cmdlet that supports this feature is `format-table`. In the above example, since `format-table` is already required to provide condensed display, we may use it instead of `select-object`. We rewrite the above script to use `format-table`. The

```

gci c:\windows -file |
  format-table name, @{label='MB';expression={$_.length / 1mb }} -
  auto

```

```

PS C:\users>
PS C:\users> gci c:\windows -file |
>>>     format-table name, @{label='MB';expression={$_.length / 1mb }} | select-object name, MB

```

Name	MB
atiogl.xml	0.0174837112426758
ativpsrm.bin	0
bfsvc.exe	0.06787109375
bootstat.dat	0.064453125
csup.txt	1.1444091796875E-05
DfsrAdmin.exe	0.24609375
DfsrAdmin.exe.config	0.00125408172607422
DirectX.log	0.0306711196899414
DPINST.LOG	0.0119705200195313
DtcInstall.log	0.00193309783935547
epplauncher.mif	0.00185489654541016
explorer.exe	2.73876953125
fveupdate.exe	0.0146484375
HelpPane.exe	0.69970703125
hh.exe	0.01611328125
IE10_main.log	0.00596809387207031
IE11_main.log	0.00723075866699219
IE9_main.log	0.00345611572265625
iis7.log	0.0280580520629883
MEMORY.DMP	722.334157943726
mib.bin	0.041132926940918
msdfmap.ini	0.00133991241455078
msxml4-KB954430-enu.LOG	0.280910491943359

We may abbreviate the key and value labels as below.

```

gci c:\windows -file |
    format-table name, @{l='MB';e={$_.length / 1mb }} -auto

```

The derived property MB is not very neat in the display. Let's round the value of MB to two decimal places by using the [math] class static method *round*:

```

gci c:\windows -file |
    format-table name, @{l='MB';e={[math]::round($_.length / 1mb,2) }} -auto

```

```

PS C:\users>
PS C:\users> gci c:\windows -file | 
>>>     format-table name, @{l='MB';e={[math]::round($_.length / 1mb,2)}}

```

Name	MB
atiogl.xml	0.02
ativpsrm.bin	0
bfsvc.exe	0.07
bootstat.dat	0.06
csup.txt	0
DfsrAdmin.exe	0.25
DfsrAdmin.exe.config	0
DirectX.log	0.03
DPINST.LOG	0.01
DtcInstall.log	0
epplauncher.mif	0
explorer.exe	2.74
fveupdate.exe	0.01
HelpPane.exe	0.7
hh.exe	0.02
IE10_main.log	0.01
IE11_main.log	0.01
IE9_main.log	0
iis7.log	0.03
MEMORY.DMP	722.33
mib.bin	0.04
msdfmap.ini	0
msxml4-KB954430-enu.LOG	0.28
msxml4-KB973688-enu.LOG	0.28

In the expression `[math]::round($_.length / 1mb,2)`, we passed the result of `$_.length / 1mb` to the `round` method. We instruct `round` to round to two decimal places by passing the value 2 as the second parameter.

We also want to optimize this a bit. Usually we're not interested in small files so let's see only files whose size is greater than 1MB (in real life this would be more like 1GB). We rewrite the pipeline to include a `where-object` to select only files with the desired sizes.

```

gci c:\windows -file | 
    where-object { $_.length -gt 1MB } |
        format-table name, @{l='MB';e={[math]::round($_.length / 1mb,2)}}
    -auto

```

```
PS C:\users>
PS C:\users> gci c:\windows -file |
>>>     where-object { $_.Length -gt 1MB } |
>>>     format-table name, @{l='MB';e={[math]::round($_.Length / 1mb,2)}
```

Name	MB
explorer.exe	2.74
MEMORY.DMP	722.33
WindowsUpdate.log	1.1

Scriptblocks may be used to create some cool synthetic properties. For example, I want see how days since last time the file was updated

```
gci -path c:\windows -file |
    format-table name, LastWriteTime,
        @{l='Last Written(days)';e={(get-date) -
        $_.LastWriteTime).Days} } -auto
```

Name	LastWriteTime	Last Written(days)
atiogl.xml	6/17/2009 6:53:12 AM	2256
ativpsrm.bin	8/16/2010 6:29:19 PM	1830
bfsvc.exe	11/20/2010 8:24:27 AM	1735
bootstat.dat	8/21/2015 11:00:42 AM	0
csup.txt	8/16/2010 7:26:41 PM	1830
DfsrAdmin.exe	7/13/2009 9:46:10 PM	2229
DfsrAdmin.exe.config	2/28/2013 10:38:48 AM	904
DirectX.log	10/22/2012 10:02:22 AM	1033
DPINST.LOG	8/16/2010 4:34:42 PM	1830
DtcInstall.log	8/16/2010 4:55:18 PM	1830
eplauncher.mif	5/13/2015 3:28:19 AM	100
explorer.exe	2/25/2011 1:19:30 AM	1638
fveupdate.exe	7/13/2009 9:39:10 PM	2229
HelpPane.exe	7/13/2009 9:39:12 PM	2229
hh.exe	7/13/2009 9:39:12 PM	2229
IE10_main.log	3/7/2013 1:56:37 PM	897
IE11_main.log	11/19/2013 3:04:57 AM	640
IE9_main.log	3/28/2011 2:23:48 PM	1607
iis7.log	11/14/2012 10:36:55 AM	1010
MEMORY.DMP	9/6/2013 3:42:23 PM	713
mib.bin	7/13/2009 7:06:54 PM	2229
msdfmap.ini	6/10/2009 4:36:48 PM	2262
msxml4-KB954430-enu.LOG	8/31/2010 11:06:17 AM	1816

Let's look at what's happening in the script block associated with `format-table` expression:

PowerShell executes `(get-date)` to retrieve the current date. The parentheses are required to tell PowerShell that the content between them is a sub-expression that must execute first.

After the current date object is retrieved, we subtract the last write time of the current fileinfo object in the pipeline from the current date object, `(get-date) - $_.LastWriteTime` resulting in a timespan object.

A timespan object has a number of date difference properties including the Days property. We enclose the whole expression to tell PowerShell we want

to reference .Days property of the timespan object created by the expression, `((get-date) - $_.LastWriteTime).Days` .

Using format-table cmdlet:

As already seen many times, the format-table cmdlet formats property values in a tabular form. This cmdlet has other parameters of interest.

Using -groupby:

This switch allows grouping a property or properties by one property. The example groups files by their extension. We use the sort-object cmdlet to sort the objects on the extension property. This is necessary because for grouping to occur the objects must be sorted on the property on which they are being grouped. Since format-table does not sort, it must be presented with objects already presorted. Another important reason is that get-childitem streams objects so format-table would receive individual objects which cannot be grouped. In addition to sorting, sort-object also blocks the pipeline accumulating all objects before passing them on to format-table.

```
get-childitem -path c:\windows -file | sort-object extension |  
format-table name -groupby extension
```

```
PS C:\users>
PS C:\users> get-childitem -path c:\windows -file | sort-object extension
>>>           format-table name -groupby extension

    Extension: .bin

Name
----
ativpsrm.bin
mib.bin

    Extension: .config

Name
----
DfsrAdmin.exe.config

    Extension: .dat

Name
----
bootstat.dat

    Extension: .dll
```

Let's look at another example where we need to list running process by publisher (company) name. The pipeline use select-object to select the properties of interest. This is required by

```
get-process | select-object company, name, description |
              sort-object company, name, description |
              format-table name, description -groupby company -
              autosize
```

```

PS C:\users>
PS C:\users> get-process | select-object company, name, description
>>>                               sort-object company, name, description | 
>>>                               format-table name, description -groupby company -a

```

Company: Intel Corporation

Name	Description
IAAnotif	Event Monitor User Notification Tool

Company: Microsoft Corp.

Name	Description
BDAppHost	BDAppHost.exe
BDExtHost	BDExtHost.exe
BDRuntimeHost	BDRuntimeHost.exe
BDSurrogateHost	BDSurrogateHost.exe
BDSurrogateHost	BDSurrogateHost.exe
BingDesktop	Bing Desktop Application

Company: Microsoft Corporation

Name	Description
conhost	Console Window Host
CSISYNCCLIENT	Microsoft Office Document Cache Sync Client Inter
dwm	Desktop Window Manager
explorer	Windows Explorer
GWX	GWX
MSOSYNC	Microsoft Office Document Cache
msseces	Microsoft Security Client User Interface

Using -view:

The -view switch parameter organizes the output by the specified property. The example below shows the processes organized by their start time. The where-object in the pipeline is required to filter the processes, which includes many services, that don't have a start time.

```
get-process | where-object { $_.starttime -gt 0 } |
```

```
sort-object starttime |  
format-table -view starttime
```

```
PS C:\Users>  
PS C:\Users> get-process | where-object { $_.starttime -gt 0 } |  
>>>         sort-object starttime |  
>>>         format-table -view starttime
```

StartTime.ToString(): 8/28/2015

ProcessName	Id	HandleCount	WorkingSet64
taskhost	1940	260	16105472
dwm	1992	198	44863488
taskeng	2040	100	6791168
explorer	1040	900	73707520
iSCSIAgent	1360	183	540672
rundll32	1392	72	6524928
GWX	3104	198	888832
IAAnotif	3320	117	7241728
Dell.ControlPoint	3336	404	57044992
BcmDeviceAndTaskS...	3364	445	40456192
msseces	3376	253	16240640
BACSTray	3420	46	5439488
MSOSYNC	3484	315	10326016
sidebar	3492	359	42033152
OneDrive	3512	625	44687360

Using Hash Tables:

Many previous examples used hash table in both select-object and format-object to either create derived properties or rename columns. Hash table may be used in format-table to create custom column labels or create derived output. The pipeline below shows the processname of all running processes and the TotalRunTime

```
get-process | format-table ProcessName,  
             @{Label="TotalRunTime"; Expression={(get-date) -  
$_.StartTime}} -autosize
```

```
PS C:\Users>
PS C:\Users> get-process | format-table ProcessName,
>>>                               @{Label="TotalRunTime"; Expression={(get-date) - $_.StartTime}}
ProcessName          TotalRunTime
-----              -----
armsvc
aticlxx
atiesrxx
BACSTray           7.02:09:12.0782974
BDAppHost           7.02:08:05.4859837
BDExtHost           7.02:08:06.1119849
BDRuntimeHost       7.02:08:03.3205801
BDSurrogateHost    7.02:07:55.7243667
BDSurrogateHost    7.02:07:55.4455663
BingDesktop         7.02:09:06.7530929
BingDesktopUpdater 7.02:08:55.5104735
CCC                 7.01:58:31.4165459
chrome              7.01:58:34.5541516
chrome              7.01:58:31.4039461
```

The hash table identifies the column Label to show as the heading. The expression subtracts the current datetime from the StartTime (start datetime) to display the total running time for the process.

Format-List

The format-list cmdlet formats the output in linear form instead of the table format used by format-table.

```
get-service | select-object -first 2 | format-list *
```

```
PS C:\Users>
PS C:\Users> get-service | select-object -first 2 | format-list *
```

Name	:	AdobeARMservice
RequiredServices	:	{}
CanPauseAndContinue	:	False
CanShutdown	:	False
CanStop	:	True
DisplayName	:	Adobe Acrobat Update Service
DependentServices	:	{}
MachineName	:	.
ServiceName	:	AdobeARMservice
ServicesDependedOn	:	{}
ServiceHandle	:	
Status	:	Running
ServiceType	:	Win32OwnProcess
Site	:	
Container	:	
Name	:	AdobeFlashPlayerUpdateSvc
RequiredServices	:	{}
CanPauseAndContinue	:	False
CanShutdown	:	False
CanStop	:	False

Typically, format-list like format-table displays a default set of properties of the object. We can stipulate specific property names to displays those properties. We can also use the wildcard * to inform the format cmdlet to display all properties of the object. The * wildcard may also be used with format-table but doesn't result in a useable display.

Format-Wide

This cmdlet provides a wide list of one property of an object. If a property name is not specified, format-wide uses a default property specific to that object. If a property name is specified, a wide display of the property occurs.

```
get-childitem c:\windows | format-wide -autosize
```

```
PS C:\Users>
PS C:\Users> get-childitem c:\windows | format-wide -autosize

    Directory: C:\windows

[addons] [AppCompat] [AppPatch]
[assembly] [Boot] [Branding]
[CSC] [Cursors] [debug]
[Dell] [diagnostics] [DigitalLocker]
[Downloaded Installations] [Downloaded Program Files] [ehome]
[en] [en-US] [Fonts]
[Globalization] [Help] [IME]
[inf] [L2Schemas] [LiveKernelRepl]
[Logs] [Media] [Microsoft.NET]
[Migration] [Minidump] [ModemLogs]
[Offline Web Pages] [Panther] [PCHEALTH]
[Performance] [PLA] [PolicyDefinition]
[Prefetch] [registration] [rescache]
[Resources] [SchCache] [schemas]
[security] [ServiceProfiles] [servicing]
[Setup] [ShellNew] [SoftwareDistr]
[Speech] [symbols] [system]
```

Basic Calculations

Tuesday, October 24, 2017
6:29 PM

Many times in scripts, the developer needs to perform some basic calculations including counting, totaling, find the maximum value, or find the minimum value. Previous sections have illustrated some examples. This page provides a more complete description

Counting the number of values in a set of values:

The algorithm for counting is very straightforward. Increment a variable every time the object passes through the pipeline.

This first example counts the number of services executing at the time the `get-service` cmdlet executes. Note that the counter `$nsvcs` is initialized to 0 before the pipeline is executed.

```
clear-host
write-host `n`n
get-service | foreach-object -begin { $nsvcs = 0} -process{ $nsvcs =
$nsvcs + 1 }
write-host "The number of running services is $nsvcs"
write-host `n`n
```

This script may be re-written more concise by using an foreach-object alias % and the increment operator ++ instead of the addition operator.

```
clear-host
write-host `n`n
$nsvcs = 0
get-service | % -begin { $nsvcs = 0} -process{ $nsvcs++ } `-
              -end { write-host "The number of running services is
$nsvcs" }
write-host `n`n
```

```
1 clear-host
2 write-host `n`n
3 $nsvcs = 0
4 get-service | % -begin { $nsvcs = 0} -process{ $nsvcs++ } `-
              -end { write-host "The number of running services is $nsvcs" }
6 write-host `n`n
```

```
The number of running services is 262
```

Generating a total of a set of values:

The algorithm for totaling (generating a sum or aggregating) a set of values is similar to that for counting but instead of incrementing a variable, we add the value, that we want to total, to the variable.

This example totals the free space on all disk drives in the computer.

```
clear-host
write-host `n`n
$totalfree = 0
get-psdrive -psprovider filesystem | foreach-object { $totalfree =
$totalfree + $_.free }
$totalfree = $totalfree/1GB
write-host "The amount of free space on all drives is $totalfree "
write-host `n`n
```

Writing this in more concise syntax again using the foreach alias, the `+
=` assignment operator, and enhanced syntax.

```
clear-host
write-host `n`n
get-psdrive -psprovider filesystem |
% -begin { $totalfree = 0 } -process { $totalfree += $_.free } -
end { $totalfree = $totalfree/1GB }
write-host "The amount of free space on all drives is $totalfree GB "
write-host `n`n
```

```
1 clear-host
2 write-host `n`n
3 get-psdrive -psprovider filesystem |
4 % -begin { $totalfree = 0 } -process { $totalfree += $_.free } -end { $totalfree =
5 write-host "The amount of free space on all drives is $totalfree GB "
6 write-host `n`n|
```

```
The amount of free space on all drives is 2748.01844024658 GB
```

Finding the maximum value in set of values:

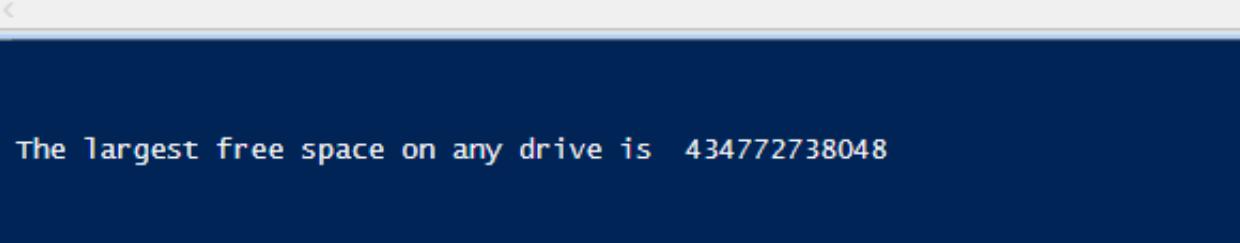
Finding the maximum value in a set of values requires a variable to hold the current maximum value. As objects pass through the pipeline, the current object

value is tested against the variable value. If the current object value is greater than the variable value, the variable value is replaced by the current object value.

This example demonstrates the principle. The script determines the largest free space over all drives installed on the computer. Note that \$maxfree, the variable to hold the largest free space is initialized to 0. As the driveinfo objects flow through the pipeline, the free space of the current object is compared against the value of \$maxfree. If the free space of the current object is not greater than, where-object will drop the object from the pipeline and the foreach-object does not execute. If the free space of the current driveinfo object is greater than the value in \$maxfree, the object is passed on to foreach-object where the value of \$maxfree is replaced by the free space of the current drive info object.

```
clear-host
write-host `n`n
$maxfree = 0
get-psdrive -psprovider filesystem |
    where-object { $_.free -gt $maxfree } |
        foreach { $maxfree = $_.free }
write-host "The largest free space on any drive is $maxfree "
write-host `n`n
```

```
1 clear-host
2 write-host `n`n
3 $maxfree = 0
4 get-psdrive -psprovider filesystem |
5     where-object { $_.free -gt $maxfree } |
6         foreach { $maxfree = $_.free }
7 write-host "The largest free space on any drive is $maxfree "
8 write-host `n`n
```



```
The largest free space on any drive is 434772738048
```

Finding the minimum value in a set of values:

Structurally the algorithm for finding the minimum is similar to find the maximum. The differences are the initial value assigned to the variable that holds the minimum value. The other difference is the comparison operator used. The algorithm requires that the current object value is compared against the variable holding the current minimum. If the current object value is less than the minimum variable value, then the minimum variable value is replaced by the current object value. Since the comparison operator is less than, the initial value assigned must be a large number.

The example below finds the smallest free space of all drives installed. Note the starting values assigned to \$minfree, 1000TB is a very large number. However, any number larger than the largest number in the set of values is sufficient.

```
clear-host
write-host `n`n
$minfree = 1000TB
get-psdrive -psprovider filesystem |
    where-object { $_.free -lt $minfree } |
        foreach { $minfree = $_.free }
write-host "The smallest free space on any drive is $minfree "
write-host `n`n
```

```
1  clear-host
2  write-host `n`n
3  $minfree = 1000TB
4  get-psdrive -psprovider filesystem |
5      where-object { $_.free -lt $minfree } |
6          foreach { $minfree = $_.free }
7  write-host "The smallest free space on any drive is $minfree "
8  write-host `n`n
```

```
The smallest free space on any drive is 507297792
```

Working with Paths

Tuesday, April 02, 2013

7:49 PM

PowerShell abstracts a number of hierarchical data repositories available in Windows. The terminology used to describe the elements of a hierarchical data store are *node* (*also sometimes referred to as a branch*) and *leaf*. A node is usually a container object which contains leaf objects. This terminology is used because if the hierarchical data store would be diagrammed it would appear as an inverted tree. An example is the file system where directories are the nodes (branches) and the files are the leafs. This is best illustrated by the legacy DOS command *tree* as shown below. Note, how this example combines legacy command with PowerShell cmdlets to produce the desired result.

```
tree /f | select-object -skip 2 | out-host -paging
```

```
PS C:\windows>
PS C:\windows> tree /f | select-object -skip 2 | out-host -paging
C:.
    bfcsvc.exe
    certenroll.log
    comsetup.log
    diagerr.xml
    diagwrn.xml
    DPINST.LOG
    DtcInstall.log
    Enterprise.xml
    explorer.exe
    HelpPane.exe
    hh.exe
    lsaservice.log
    mib.bin
    NtGPopKeySrv.log
    notepad.exe
    PFR0.log
    regedit.exe
    setupact.log
    setuperr.log
    splwow64.exe
    system.ini
    twain_32.dll
    win.ini
    WindowsUpdate.log
    winhlp32.exe
    WMSysPr9.prx
    write.exe

    addins
        FXSEXT.ecf

    ADFS
        ar
        bg
        cs
        da
        de
        el
        en
        en-GB
```

Having understood the shortcomings of the legacy cmd shell, the PowerShell developers provided a number of cmdlets are available for working with paths. These cmdlets are intended to be used in scripts to dynamically build paths.

The *resolve-path* cmdlet provides an absolute path from the relative path provided.

```
resolve-path ..\windows\system32..\system..\system32..\..\..
resolve-path
\windows\system32\drivers\etc\..\..\..\system..\media\..\..\logs\..\..
```

```
PS C:\users>
PS C:\users> resolve-path ..\windows\system32..\system..\system32..\..
Path
-----
C:\

PS C:\users> resolve-path \windows\system32\drivers\etc\..\..\..\system..\media\..\..\log
Path
-----
C:\

PS C:\users> resolve-path \windows\system32\drivers\etc\..\..\..\system..\media\..\..\log\*
Path
-----
C:\windows\media\Alarm01.wav
C:\windows\media\Alarm02.wav
C:\windows\media\Alarm03.wav
C:\windows\media\Alarm04.wav
C:\windows\media\Alarm05.wav
C:\windows\media\Alarm06.wav
C:\windows\media\Alarm07.wav
C:\windows\media\Alarm08.wav
```

The *split-path* cmdlet may be used to return the parent directory, child directory, or the filename in a path. In the last example of split-path in the illustration below, the *-resolve* switch parameter causes split-path to display the files, as determined by the wildcard, contained within the path.

```
PS C:\users>
PS C:\users> $path = 'c:\windows\system32\drivers\etc\hosts'
PS C:\users>
PS C:\users> split-path -path $path -leaf
hosts
PS C:\users>
PS C:\users> split-path -path $path -parent
c:\windows\system32\drivers\etc
PS C:\users>
PS C:\users> split-path -path 'c:\windows\system32\drivers\etc\*' -leaf
hosts
\lmhosts.sam
networks
protocol
services
PS C:\users>
```

The *join-path* cmdlet combines a parent path and a child path paths into one path. A child path means a path that is contained in the parent path. In the last example, the *-resolve* switch parameter resolves the files specified by the wildcard.

```
PS C:\users>
PS C:\users> join-path -path 'c:\windows\system32' -childpath 'drivers\etc'
c:\windows\system32\drivers\etc
PS C:\users>
PS C:\users> join-path -path 'c:\windows\system32' -childpath 'drivers\etc\*'
C:\windows\system32\drivers\etc\hosts
C:\windows\system32\drivers\etc\lmhosts.sam
C:\windows\system32\drivers\etc\networks
C:\windows\system32\drivers\etc\protocol
C:\windows\system32\drivers\etc\services
PS C:\users>
```

Navigating the FileSystem

Tuesday, April 2, 2013
4:53 PM

The *Set-Location* cmdlet is used to navigate all hierarchical data stores like the file system. When constructing path names for the file system or other abstracted

hierarchical repositories, metacharacters may be used to provider shorter and more efficient paths.

Metachatacter	Meaning
.	Current directory
..	Parent directory
\	Root
~	Home directory

Metacharacters may be used with set-location and other cmdlets to specify absolute and relative pathnames for files and directories for the file system and nodes and leafs for the Registry, Active Directory, and other hierarchical repositories.

Borrowing from Linux shells, PowerShell provides a convenient method for "going home".

```
set-location ~  
set-location $home
```

Navigating to the parent directory and to the root requires the identical path used in the legacy cmd shell or any Linux shell with the exception that Linux uses a *forward slash* instead of a *back slash*.

```
PS C:\Users>
PS C:\Users> set-location ..
PS C:>
PS C:> set-location users
PS C:\users>
PS C:\users> set-location \
PS C:>
```

The location cmdlets may be used to create a location memory within a script or within the console. This is accomplished by storing a location in a variable using the *get-location* cmdlet. This cmdlet stores the current path as a PathInfo object in the variable. These variables may then be used with *set-location* to navigate to the locations specified in the variable.

```
PS C:\users>
PS C:\users> $usersLoc = get-location
PS C:\users>
PS C:\users> set-location c:\windows\system32
PS C:\windows\system32>
PS C:\windows\system32> $sys32Loc = get-location
PS C:\windows\system32>
PS C:\windows\system32> set-location drivers\etc
PS C:\windows\system32\drivers\etc>
PS C:\windows\system32\drivers\etc> $etcLoc = get-location
PS C:\windows\system32\drivers\etc>
PS C:\windows\system32\drivers\etc> set-location ..\..\..
PS C:\windows>
PS C:\windows> $winLoc = get-location
PS C:\windows>
PS C:\windows> set-location $usersLoc
PS C:\users>
PS C:\users> set-location $etcLoc
PS C:\windows\system32\drivers\etc>
PS C:\windows\system32\drivers\etc> set-location $winLoc
PS C:\windows>
PS C:\windows> set-location $usersLoc
PS C:\users>
```

Another method of create a location memory is borrowed from the legacy cmd shell commands *pushd* and *popd*. PowerShell implements the *push-location* and *pop-location* cmdlets to create a stack of locations that may be pushed and popped (i.e., navigated to).

```
PS C:\Users>
PS C:\Users> push-location      # put this location at the top of the stack
PS C:\Users>
PS C:\Users> set-location c:\windows
PS C:\Windows>
PS C:\Windows> push-location  # push the locations in the stack down and put this location at the top
PS C:\Windows>
PS C:\Windows> set-location .\system32\drivers\etc
PS C:\Windows\system32\drivers\etc>
PS C:\Windows\system32\drivers\etc> push-location  # push locations down and place this location at the top
PS C:\Windows\system32\drivers\etc>
PS C:\Windows\system32\drivers\etc> pop-location  # pop the location off the top and navigate to it
PS C:\Windows\system32\drivers\etc>
PS C:\Windows\system32\drivers\etc> # This was the location at the top so we didn't go anywhere
PS C:\Windows\system32\drivers\etc>
PS C:\Windows\system32\drivers\etc> pop-location  # this should navigate to the location
PS C:\Windows>
PS C:\Windows> pop-location  # this will navigate back to c:\users
PS C:\Users>
PS C:\Users>
```

Working with File System Objects

Tuesday, April 02, 2013
8:15 PM

There are a number of cmdlets for working with the file system. The two cmdlets that may be used to retrieve information about file system objects are get-childitem and get-item. When working with files get-childitem and get-item return an identical object. When working with directories, get-item must be used since get-childitem would retrieve fileInfo objects for the files contained in the directory. Let's examine some useful examples of working with file system objects.

One of the more useful applications is to find files created today or on a specific day. The inclination to find files created today is to do something like that below.

```
$now = get-date
get-childitem -recurse | where-object { $_.CreationTime -eq $now }
```

However this doesn't work since both the CreationTime property and object returned by get-date are of the [datetime] type which include both a date and a time. The CreationTime property contains the date and time the file was created. While get-date returns the current date and time. Both objects would never be equal.

Ideally the approach is to find files created after 12:00 AM today. We need a [datetime] object that reflects 12:00 AM today. We would then look for any files whose CreationTime is greater than that object. Inspecting the properties and methods of get-date, we notice the property Date which provides the required date-time.

```
$today = get-date  
$today.date
```

```
PS C:\Users>  
PS C:\Users> $today = get-date  
PS C:\Users>  
PS C:\Users> $today  
  
Thursday, September 10, 2015 3:21:39 PM  
  
PS C:\Users> $today.date  
  
Thursday, September 10, 2015 12:00:00 AM
```

Given the above, we may now write the script below that retrieves files created today.

```
$today = get-date  
get-childitem -recurse | where-object { $_.CreationTime -ge  
$today.date }
```

The above may be reduced further by using a subexpression and eliminating the \$today variable as shown below.

```
get-childitem -recurse | where-object { $_.CreationTime -ge (get-  
date).date }
```

The cmdlet *get-item* is used when working with directories or files. When used with files or directories, this cmdlet returns the identical object as *get-childitem*.

```
PS C:\Users>
PS C:\Users> $f = get-item .\temp.txt
PS C:\Users>
PS C:\Users> $f | get-member
```

TypeName: System.IO.FileInfo

Name	MemberType	Definition
---	-----	-----
Mode	CodeProperty	System.String Mode{get=Mode; }
AppendText	Method	System.IO.StreamWriter AppendText()
CopyTo	Method	System.IO.FileInfo CopyTo(string des
Create	Method	System.IO.FileStream Create()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef Create
CreateText	Method	System.IO.StreamWriter CreateText()

When *get-item* is used with a directory, a *DirectoryInfo* object is returned.

```
PS C:\Users>
PS C:\Users> $d = get-item .\Public\
PS C:\Users>
PS C:\Users> $d | gm
```

TypeName: System.IO.DirectoryInfo

Name	MemberType	Definition
---	-----	-----
Mode	CodeProperty	System.String Mode{get=Mode; }
Create	Method	void Create(), void Create(Sys
CreateObjRef	Method	System.Runtime.Remoting.ObjRef
CreateSubdirectory	Method	System.IO.DirectoryInfo Create
Delete	Method	void Delete(), void Delete(boo
EnumerateDirectories	Method	System.Collections.Generic.IEn
EnumerateFiles	Method	System.Collections.Generic.IEn
EnumerateFileSystemInfos	Method	System.Collections.Generic.IEn
Equals	Method	bool Equals(System.Object obj)

When working with FileInfo and DirectoryInfo objects we must always be aware that the object exists in memory and the file or directory exists on the file system which produces some interesting phenomena. Let's look at the scenario below.

```
$f = get-item .\temp.txt
$f.Exists
$f.Delete()      # delete the file on the file system
$f.Exists        # PowerShell still thinks the file exists
$f.Refresh()     # refresh the object in memory
$f.Exists        # because the object was refreshed, the correct state
is shown
```

```
PS C:\Users>
PS C:\Users> $f = get-item .\temp.txt
PS C:\Users>
PS C:\Users> $f.Exists
True
PS C:\Users> $f.Delete()      # delete the file on the file system
PS C:\Users>
PS C:\Users> $f.Exists        # PowerShell still thinks the file exists
True
PS C:\Users>
PS C:\Users> $f.Refresh()     # refresh the object in memory
PS C:\Users>
PS C:\Users> $f.Exists        # because the object was refreshed, the correct state is shown
False
PS C:\Users>
```

File system objects are not dynamically updated. So, if the underlying data structure (file or directory) is deleted, moved, or renamed. The PowerShell object residing in memory is not aware of the change.

Creating a directory:

The *new-item* cmdlet may be used to create directories and files. While new-item is the preferred (and maybe the only method) for creating directories, many cmdlets exist for creating files as shown in the next page. Let's look at a couple of examples.

The example below creates a directory called *mytemp* in the current directory (-path .)

```

PS C:\Users>
PS C:\Users> new-item -path . -name mytemp -itemtype directory

    Directory: C:\Users

Mode          LastWriteTime         Length Name
----          -----          ----- 
d----  9/14/2015 4:19 PM           0    mytemp

```

In the next example the directory *downloads* is created in that path *c:\dell*.

```

PS C:\Users>
PS C:\Users> new-item -path c:\dell -name downloads -itemtype directory

    Directory: C:\dell

Mode          LastWriteTime         Length Name
----          -----          ----- 
d----  9/14/2015 4:21 PM           0    downloads

```

Removing files and directories:

The *remove-item* cmdlet is used to remove a file or directory. Using the *-recurse* parameter causes the cmdlet to remove files in the target directory and also all subdirectories and files.

```

PS C:\users>
PS C:\users> remove-item file.txt
PS C:\users>
PS C:\users> remove-item .\mydir\ -recurse
PS C:\users>

```

Renaming file and directories:

The *rename-item* cmdlet renames files and directories.

Working with ACLs:

Once we have the file system object instantiated, we may use methods to get or set access control list (ACL) properties.

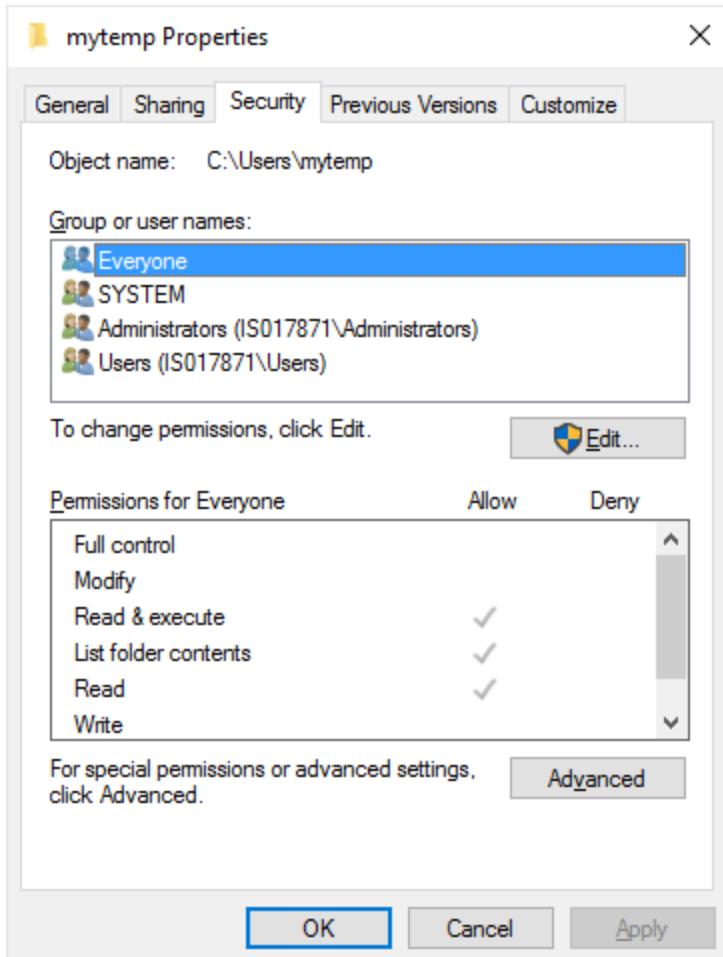
In this example, we create a directory on the file system and save the resultant DirectoryInfo object in the variable \$d. When then use the *GetAccessControl* method to retrieve the ACL properties and pipe the properties into *format-list*.

```
$d = new-item -itemType directory -path .\mytemp  
$d.GetAccessControl() | format-list *
```

```
PS C:\Users>  
PS C:\Users> $d = new-item -itemType directory -path .\mytemp  
PS C:\Users>  
PS C:\Users> $d.GetAccessControl() | format-list *
```


AccessToString	:	NT AUTHORITY\SYSTEM Allow FullControl BUILTIN\Administrators Allow FullControl BUILTIN\Users Allow ReadAndExecute, Synchronize BUILTIN\Users Allow -1610612736 Everyone Allow ReadAndExecute, Synchronize Everyone Allow -1610612736
AuditToString	:	
Path	:	
Owner	:	BUILTIN\Administrators
Group	:	IS017871\mgale
Access	:	{System.Security.AccessControl.FileSystemAccessRule, System.Security.AccessControl.FileSystemAccessRule, System.Security.AccessControl.FileSystemAccessRule, System.Security.AccessControl.FileSystemAccessRule}
Sddl	:	O:BAG:S-1-5-21-849909216-2755930895-1294993 ;ID;0x1200a9;;BU)(A;OICII0ID;GXGR;;;BU)(A;OICII0ID;GXGR;;;BU)
AccessRightType	:	System.Security.AccessControl.FileSystemRights
AccessRuleType	:	System.Security.AccessControl.FileSystemAccessRule
AuditRuleType	:	System.Security.AccessControl.FileSystemAuditRule
AreAccessRulesProtected	:	False
AreAuditRulesProtected	:	False
AreAccessRulesCanonical	:	True
AreAuditRulesCanonical	:	True

Using the GUI to inspect the Security properties of the new-created directory, we see the following for the Everyone group.



We next we use the script below to change the ACL. In this script we do the following:

1. Set variables to hold the properties of the new ACL rule.
2. Create the ACL rule and store the resultant rule object in \$rule
3. Get the security descriptor (ACL) and store it in the variable \$acl.
4. Display the security descriptor on the console
5. Use the AddAccessRule method of the ACL object to add the rule to the ACL object.

6. Use set-acl to update the security descriptor for the directory.
 7. Display the updated security descriptor on the console.

```
$group = "Everyone"
$control = "FullControl"
$permit = "Allow"
$rule =
    New-Object System.Security.AccessControl.FileSystemAccessRule
$group,$control,$permit

$acl = get-acl .\mytemp
$acl

$acl.AddAccessRule($rule)
set-acl -path .\mytemp -Aclobj $acl

get-acl .\mytemp
```

```
PS C:\users>
PS C:\users> $group = "Everyone"
PS C:\users> $control = "FullControl"
PS C:\users> $permit = "Allow"
PS C:\users> $rule =
>>>     New-Object System.Security.AccessControl.FileSystemAccessRule $group,$control,
PS C:\users>
PS C:\users> $acl = get-acl .\mytemp
PS C:\users> $acl

    Directory: C:\users

Path          Owner          Access
----          ----          -----
mytemp BUILTIN\Administrators NT AUTHORITY\SYSTEM Allow FullControl...
PS C:\users>
PS C:\users> $acl.AddAccessRule($rule)
PS C:\users> set-acl -path .\mytemp -Aclobj $acl
PS C:\users>
PS C:\users>
PS C:\users> get-acl .\mytemp

    Directory: C:\users

Path          Owner          Access
----          ----          -----
mytemp BUILTIN\Administrators Everyone Allow FullControl...
```

Working with Files

Thursday, January 16, 2014
11:45 AM

Writing to text files:

There are two cmdlets for writing text files: *set-content* and *out-file*. There are some very important differences between the two. The first important difference is encoding. The *set-content* cmdlet, by default, creates a text file with ASCII encoding while *out-file* creates, by default, a Unicode UTF-16 encoded file. This example illustrates this difference .

```
'abcdef' | set-content sc.txt
'abcdef' | out-file ot.txt
get-content sc.txt
get-content ot.txt
```

```
PS C:\Users>
PS C:\Users> 'abcdef' | set-content sc.txt
PS C:\Users>
PS C:\Users> 'abcdef' | out-file ot.txt
PS C:\Users>
PS C:\Users> get-content sc.txt
abcdef
PS C:\Users>
PS C:\Users> get-content ot.txt
abcdef
```

The illustrations appears to show that there are not difference between the files. Let's inspect the files using the WinHex editor.

From the illustration below sc.txt created by *set-content* appears to be a typical ASCII encoded file. The string 'abcdef' is stored as ASCII 6162636466566. The last two bytes are the carriage return and line feed characters which indicate the end of the line, i.e., new-line.

In the file created by the `out-file` cmdlet, the first two bytes of the file are FFFE which are byte order marks and indicate that this file is encoded in UTF-16 and little endian. [https://msdn.microsoft.com/en-us/library/windows/desktop/dd374101\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd374101(v=vs.85).aspx)

UTF-16 means that two bytes are used to represent a character, note the additional byte for every character. Endianness indicates the byte order. Little endian means the least significant byte is stored in the lowest memory location. In this case, the string 'abcdef' is stored as the ASCII Unicode 610062006300640065006600. In this example, the character "a" is stored as "6100" a two byte code where "00" is the most significant byte.

The `-Encoding` parameter may be used with `out-file` and with `set-content` to change the default encoding to that specified by the parameter.

```
PS C:\Users> PS C:\Users> 'abcdef' | out-file ot-ascii.txt -encoding ascii  
PS C:\Users> 'abcdef' | out-file ot-bigendian.txt -encoding bigendianunicode  
PS C:\Users>
```

In the first execution a pure ASCII file is created as shown below. This file is identical to that created by *set-content*.

For the second execution of out-file, a big endian file is created which means that the most significant byte is stored in the lowest memory location.

In this illustration note the byte order mark is the reverse of the little endian file created earlier. The characters are stored in the big endian ASCII Unicode. So, the character "a" is stored as "0061" where, again, "00" is the most significant byte.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	FE	FF	00	61	00	62	00	63	00	64	00	65	00	66	00	0D
00000010	00	0A														

For more information on endianness, see
<https://en.wikipedia.org/wiki/Endianness>.

The out-file cmdlet writes whatever is presented via the pipeline; so the file contains what would normally be displayed on the console. The *set-content* cmdlet is more selective as to the content written to a file.

```
PS C:\Users>
PS C:\Users> get-childitem | select-object -last 3 | out-file -filepath ot-dirlist.txt -encoding Ascii
PS C:\Users>
PS C:\Users> get-childitem | select-object -last 3 | set-content sc-dirlist.txt
PS C:\Users>
PS C:\Users> get-content .\ot-dirlist.txt

Directory: C:\Users

Mode                LastWriteTime         Length Name
----                -----          ----  --
-a----   9/15/2015 1:26 PM            8    sc.txt
-a----   8/21/2015 11:23 AM          78    servers.csv
-a----   8/7/2015  1:44 PM        143    users.csv

PS C:\Users> get-content .\sc-dirlist.txt
sc.txt
servers.csv
users.csv
```

The *set-content* cmdlet may be used to write hex data to a file. In the example below, we write the string 'abcdef' represented as hex to the file sc.txt using byte encoding. We first create the byte array \$btext containing the string.

```
[byte[]] $btext = 0x61,0x62,0x63,0x64,0x65,0x66  
set-content sc.txt -value $btext -encoding byte  
get-content sc.txt  
get-content sc.txt -encoding byte
```

```
PS C:\Users>  
PS C:\Users> [byte[]] $btext = 0x61,0x62,0x63,0x64,0x65,0x66  
PS C:\Users> set-content sc.txt -value $btext -encoding byte  
PS C:\Users> get-content sc.txt  
abcdef  
PS C:\Users> get-content sc.txt -encoding byte  
97  
98  
99  
100  
101  
102
```

Note, 97_{10} has the value of 61_{16} ; 98_{10} has the value of 62_{16} ; etc.

The *add-content* cmdlet is available for appending content to existing files. This cmdlet shares many of the parameters available in *set-content*. In the example below, we add a new entry to the hosts file.

```
$path = resolve-path c:\win*\sys*\dr*\etc  
"198.111.176.6 wcc1" | add-content (join-path -path $path -  
childpath 'hosts')
```

Reading from text files:

The *get-content* cmdlet is used to read the contents of a file. This cmdlet reads the whole file into memory so exercise caution. The *-totalcount* parameter limits the number of lines retrieved from the file. Correspondingly, the *-tail* retrieves the last n lines from the file.

```
get-content c:\windows\WindowsUpdate.log | out-host -paging  
get-content c:\windows\WindowsUpdate.log -totalcount 10  
get-content c:\windows\WindowsUpdate.log -tail 10
```

```

PS C:\Users>
PS C:\Users> get-content C:\windows\DirectX.log -TotalCount 10
02/20/15 22:37:09: dsetup32: IsWow64(): running on Wow64.
02/20/15 22:37:11: DSETUP: DirectXSetupA(): hWnd: 00130058 dwFlags: 000100

02/20/15 22:37:11: dsetup32: == SetupForDirectX() start ==
02/20/15 22:37:11: dsetup32: May 31 2007 19:07:43
02/20/15 22:37:11: dsetup32: SetupForDirectX(): query dxsetup command: res
02/20/15 22:37:11: dsetup32: DXSetupCommand = 0.
02/20/15 22:37:11: dsetup32: IsIA64(): not IA64.
PS C:\Users>
PS C:\Users> get-content C:\windows\DirectX.log -Tail 10
04/23/15 17:53:45: infinst: Installing d:\usrtemp\DX86D6.tmp\xact3_0_x64.i
04/23/15 17:53:45: infinst: Target file: 'C:\Windows\system32\xactengine3_0
    Target file is Version 9.22.1284.0
    Source file is Version 9.22.1284.0
04/23/15 17:53:45: infinst: Currently C:\Windows\system32\xactengine3_0.dll
04/23/15 17:53:45: infinst: Installing d:\usrtemp\DX86D6.tmp\xaudio2_0_x64.i
04/23/15 17:53:45: infinst: Target file: 'C:\Windows\system32\xaudio2_0.dll
    Target file is Version 9.22.1284.0
    Source file is Version 9.22.1284.0
04/23/15 17:53:45: infinst: Currently C:\Windows\system32\xaudio2_0.dll is

```

The Magic Number of a file:

The magic number of a file in an Operating Systems is a variable length byte string that identifies the file type. See

[http://en.wikipedia.org/wiki/Magic_number_\(programming\)](http://en.wikipedia.org/wiki/Magic_number_(programming)) for more information.

The following example reads the magic number of notepad.exe found in the c:\windows directory and displays the magic number. The get-content cmdlet is used with encodingof byte which means the file is read as a series of bytes. The -total parameter indicates the number of bytes to read. The get-content cmdlet creates a four byte array which is stored in \$mn. Then the script converts each entry of the \$mn array, i.e., each byte, into a character which are then stored in \$code. Displaying \$code on the console shows "MZ" which indicates that it is an executable. The magic number in this case are "the initials of the designer of the executable file format, Mark Zbikowski".

```

$mn = get-content c:\windows\notepad.exe -encoding byte -total 4
$code = [char] $mn[0] + [char] $mn[1] + [char] $mn[2] + [char] $mn[3]
write-host $code

```

```
PS C:\Users>
PS C:\Users> $mn = get-content c:\windows\notepad.exe -encoding byte -total
PS C:\Users> $code = [char] $mn[0] + [char] $mn[1] + [char] $mn[2] + [char]
PS C:\Users> write-host $code
MZ
```

The next example is similar to the previous. In this case, we get the magic number of a jpg and use the format operator "-f" to display the code as hexadecimal. The syntax "{0:x}" is interpreted as first variable (0) to the right of "-f" must be displayed as hexadecimal (x). Similarly, "{1:x}" is interpreted as second variable (1) to the right of "-f" must be displayed as hexadecimal (x).

```
$mn = get-content pic.jpg -encoding byte -total 4
$code = [char] $mn[0] + [char] $mn[1] + [char] $mn[2] + [char] $mn[3]
"{0:x} {1:x} {2:x} {3:x}" -f $mn[0], $mn[1], $mn[2], $mn[3]
```

```
PS C:\users>
PS C:\users> $mn = get-content pic.jpg -encoding byte -total 4
PS C:\users> $code = [char] $mn[0] + [char] $mn[1] + [char] $mn[2] + [char] $mn[3]
PS C:\users> "{0:x} {1:x} {2:x} {3:x}" -f $mn[0], $mn[1], $mn[2], $mn[3]
ff d8 ff e1
```

Alternate Data Streams

Tuesday, April 02, 2013
8:15 PM

One of the features of the NTFS file system is Alternate Data Streams.

"I don't mean to be smart aleck about it...but that's what it is. We know from my older blog that a file is divided up into 'attributes' and one of these attributes is \$DATA or simply called the data attribute. It is the part of the file we put data into. So if I have a text file that says, "This is my text", then if I look at the data attribute, it will contain a stream of data that reads, "This is my text". However, this is the normal data stream, sometimes called the primary data stream, but more accurately it is called the unnamed data stream"

From <<http://blogs.technet.com/b/askcore/archive/2013/03/24/alternate-data-streams-in-ntfs.aspx>>

As indicated above, the contents of a file in NTFS are stored as a unnamed stream in an attribute called \$DATA. NTFS supports additional named streams for a file. To demonstrate, let's create a text file with some contents in the primary data stream.

In this script we create a text file with text in the \$DATA attribute of the file. We view the contents using get-content.

```
"visible contents" | set-content ads.txt  
get-content ads.txt
```

```
PS C:\users>  
PS C:\users> "visible contents" | set-content ads.txt  
PS C:\users>  
PS C:\users> get-content ads.txt  
visible contents
```

We can now add data to an alternate stream of this file. We will create a data stream called "hide1" and store text in that stream. As shown below, using get-content to view the file only reveals the primary data stream, i.e., the normal or visible contents of the file. To view the alternate data stream hide1, we need to use the -stream parameter.

```
"hidden in stream hide1" | set-content ads.txt -stream hide1  
get-content ads.txt  
get-content ads.txt -stream hide1
```

```
PS C:\users>  
PS C:\users> "hidden in stream hide1" | set-content ads.txt -stream hide1  
PS C:\users>  
PS C:\users> get-content ads.txt  
visible contents  
PS C:\users>  
PS C:\users> get-content ads.txt -stream hide1  
hidden in stream hide1  
PS C:\users>
```

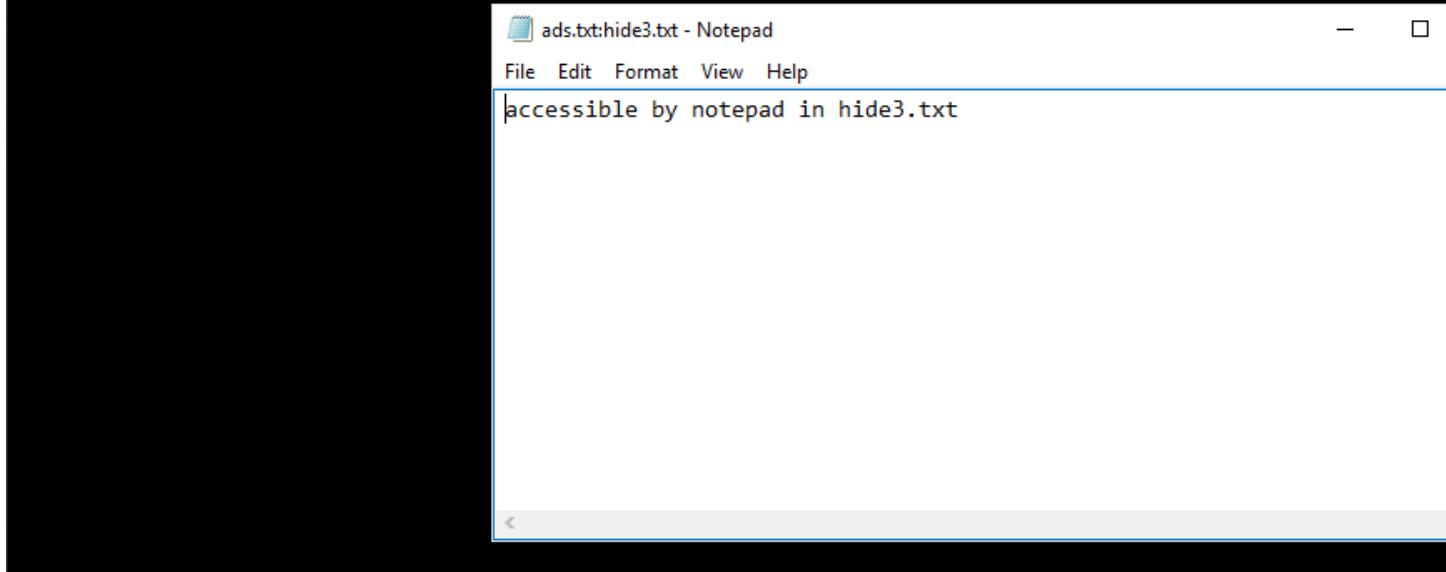
We add another alternate data stream, hide2 and use get-item to show all streams.

```
"hidden in stream hide2" | set-content ads.txt -stream hide2  
get-item ads.txt -stream *
```

```
PS C:\Users>  
PS C:\Users> "hidden in stream hide2" | set-content ads.txt -stream  
PS C:\Users>  
PS C:\Users> get-item ads.txt -stream *  
  
FileName: C:\Users\ads.txt  
  
Stream          Length  
----          ----  
:$DATA          18  
hide1           24  
hide2           24
```

Notepad is one of few Windows apps that support ADS. However, notepad is looking for a .txt extension for the data stream. So, the above data streams are not accessible through notepad. We may add an additional alternate stream with the appropriate extension using either set-content or notepad. When using notepad, a .txt extension should be added to the stream name.

```
PS C:\Users>
PS C:\Users> "accessible by notepad in hide3.txt" | set-content ads.txt -stream hide3.
PS C:\Users>
PS C:\Users> notepad ads.txt:hide3.txt
PS C:\Users>
```



Files stored in alternate data streams are not limited to text files. Any file type may be stored in an alternate data stream. In the example below, we store a jpg image in an alternate stream of a Word document.

1. Inspect the Word document. Note the file size of 12.7 KB.

PS C:\Users>
PS C:\Users> get-childitem letter.docx

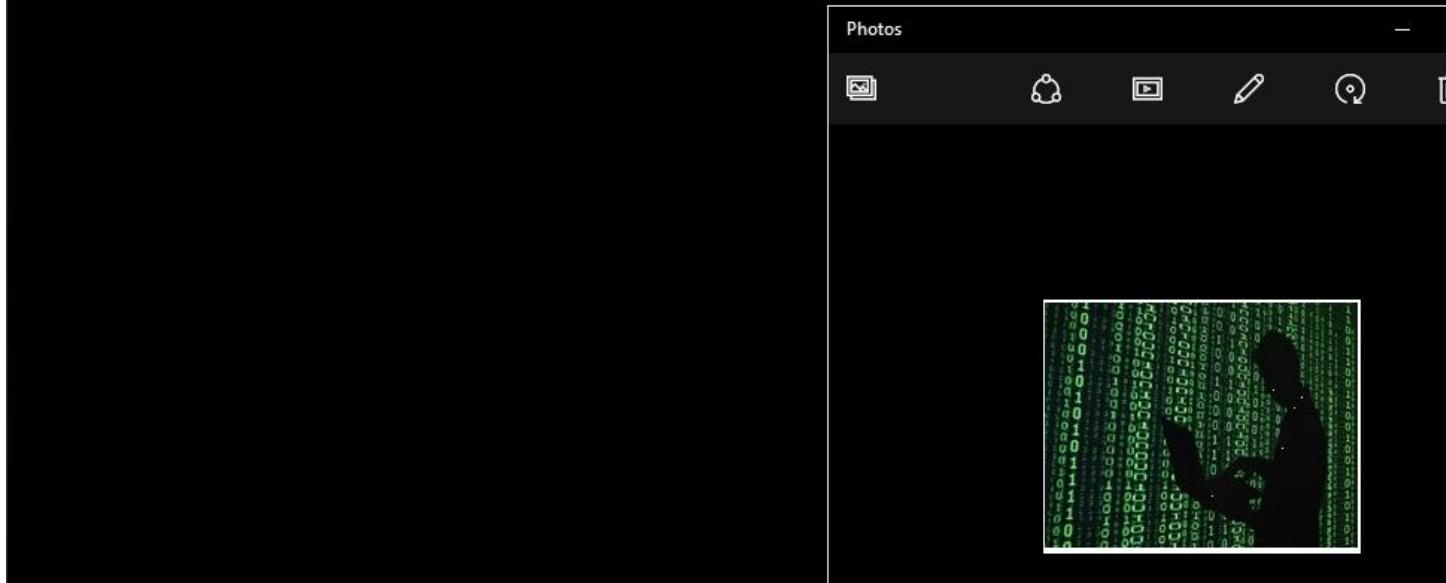
Directory: C:\Users		Length	Name
Mode	LastWriteTime		
-a---	9/16/2015 1:57 PM	12713	letter.docx

PS C:\Users> Invoke-Expression .\Letter.docx
PS C:\Users>

The Microsoft Word ribbon is visible at the top, showing tabs for FILE, HOME, INSERT, DESIGN, and PAGE. The HOME tab is selected. The font is set to Calibri (Body) at 11pt. The text color is black. The text in the document body is: 'Lorem ipsum dolor sit amet, eum in inimico qui ea ornatus partiendo, iudico offendit. Delenit laboramus nec no, sea mutat perpe ea.'

2. Inspect the image to be stored in the alternate data stream.

```
PS C:\Users>
PS C:\Users> Invoke-Expression .\anon.jpg
PS C:\Users>
```



3. Use *get-content* and *set-content* to store the image. Note, for a jpg the encoding must be byte.
4. Note that the file size hasn't changed after the image is stored.
5. We then reverse the process to retrieve the jpg in the alternate data stream and save it as new.jpg.

```
PS C:\Users>
PS C:\Users> get-content .\anon.jpg -encoding byte | set-content letter.docx -stream hiddenpic -e
PS C:\Users>
PS C:\Users> get-childitem .\Letter.docx

    Directory: C:\Users

Mode                LastWriteTime         Length Name
----                -----          -----      -----
PS C:\Users> 9/16/2015   2:02 PM           12713 Letter.docx

PS C:\Users> get-content letter.docx -stream hiddenpic -encoding byte | set-content new.jpg -encod
PS C:\Users>
PS C:\Users> invoke-expression .\new.jpg
PS C:\Users>
```



The screenshot shows a Windows File Explorer window. In the center, there is a single file named "Letter.docx". To the left of the file, it says "Mode", "LastWriteTime", "Length", and "Name". The "Length" column shows the value "12713". The "Name" column shows "Letter.docx". Above the file, there is some PowerShell command history. On the right side of the window, there is a sidebar labeled "Photos" which contains a thumbnail of the "Letter.docx" file.

Sorting

Monday, January 28, 2013
1:43 PM

The *sort-object* cmdlet is used for sorting objects on specific properties. By default this cmdlet sorts on some reasonable property of the object. If the case of file system object, *sort-object* sorts the objects on the *name* property.

```
# sort on object name  
get-childitem | sort-object | out-host -paging
```

```
PS C:\Users>  
PS C:\Users> get-childitem c:\windows | sort-object | out-host -paging  
  
Directory: C:\windows  
  
Mode LastWriteTime Length Name  
---- <----- ----- ----  
d---- 8/13/2015 4:59 AM addins  
d---- 8/27/2015 5:43 PM ADFS  
d---- 8/31/2015 3:48 PM appcompat  
d---- 8/13/2015 3:33 PM AppPatch  
d---- 9/16/2015 11:35 AM AppReadiness  
d-r--- 9/9/2015 1:15 PM assembly  
-a---l 8/13/2015 4:54 AM 61952 bfsvc.exe  
d---- 8/13/2015 4:59 AM Boot  
-a--s- 9/16/2015 10:58 AM 67584 bootstat.dat  
d---- 8/13/2015 4:59 AM Branding  
d---- 8/27/2015 9:24 PM CbsTemp  
-a---- 5/27/2015 12:24 PM 17310 certenroll.log  
-a---- 8/27/2015 5:55 PM 59396 comsetup.log  
d---- 1/26/2015 3:00 PM CSC  
d---- 8/13/2015 4:59 AM Cursors  
d---- 8/27/2015 6:04 PM debug  
d---- 1/26/2015 3:19 PM Dell  
d-r--- 8/13/2015 4:59 AM DesktopTileResources  
d-r--- 8/31/2015 3:52 PM DevicesFlow  
-a---- 8/27/2015 5:58 PM 26673 diagerr.xml  
d---- 8/13/2015 4:59 AM diagnostics  
-a---- 8/27/2015 5:58 PM 26673 diagwrn.xml  
d---- 8/13/2015 3:33 PM DigitalLocker  
<SPACE> next page; <CR> next line; Q quit  
d---s- 8/27/2015 5:43 PM Downloaded Program File  
<SPACE> next page; <CR> next line; Q quit
```

One of the more useful features of this cmdlet is to find the *big* files on the file system as shown below. By doing a descending sort on length, big files appear first.

```
get-childitem c:\windows -recurse -erroraction silentlycontinue |  
select-object name, length |  
sort-object -property length -descending |  
format-table length, name | out-host -paging
```

```
PS C:\Users>
PS C:\Users> get-childitem c:\windows -recurse -erroraction silentlycontinue | 
>>> select-object name, length | % sort-object -property length -descending | 
>>> format-table length, name | out-host -paging

Length Name
-----
2766743952 95D1590A3D928590B26B1FF4C62681332B1062D7325DE576F6272AD4E3082BA5.default_e
187424611 74782FFB16A68808483C4A24FBD3725FC6DA1788C9EED71FF684AA91B404BED8.default_e
134753440 MRT.exe
116041496 MpAvBase.vdm
95748580 5c7e961801a9c1ff04c0555fc347e6b43f6294e4
88182096 outlook-x-none.cab
86067504 MigLog.xml
76283904 SOFTWARE
76140544 SOFTWARE
52199996 IMTCT.IMG
52199996 IMTCT.IMG
51904512 DataStore.edb
48412215 setupact.log
46214656 imageres.dll
46214656 imageres.dll
46214656 imageres.dll
46214656 imageres.dll
42229760 OBJECTS.DATA
36791212 mingliub.ttc
36791212 mingliub.ttc
36110336 catdb
32941056 System.Management.Automation.ni.dll
```

We may alter the above script find *processor hogs*, i.e., process that are using most amount of CPU.

```
get-process |
    select-object name, path, cpu |
    sort-object -property cpu -descending |
    format-table cpu, name, path
```

```

PS C:\Users>
PS C:\Users> get-process | 
>>> select-object name, path, cpu | . sort-object -property cpu -descending
>>> format-table cpu, name, path

    CPU Name                               Path
    --  -- 
3872.015625 System                         C:\WINDOWS\system32\svchost.exe
  3446.6875 MsMpEng                        C:\WINDOWS\system32\svchost.exe
1550.265625 svchost                        C:\WINDOWS\system32\svchost.exe
1327.546875 svchost                        C:\WINDOWS\system32\svchost.exe
1188.078125 svchost                        C:\WINDOWS\System32\dwm.exe
  771.84375 dwm                           C:\Program Files (x86)\Microsoft Office\Off
  543.03125 OUTLOOK                        C:\Program Files (x86)\Mozilla Firefox\fire
  501.234375 firefox                         C:\Program Files (x86)\Google\Chrome\Appli
  490.484375 chrome                         C:\Windows\SystemApps\Microsoft.MicrosoftEdgeCP
  387.171875 MicrosoftEdgeCP                C:\WINDOWS\system32\SearchIndexer.exe
  359.296875 SearchIndexer                  C:\Program Files (x86)\Google\Chrome\Appli
  344.046875 chrome                         C:\Program Files (x86)\Google\Chrome\Appli
  342.75 chrome                           C:\Program Files (x86)\Google\Chrome\Appli
  336.78125 chrome                         C:\Program Files (x86)\Google\Chrome\Appli
  314.78125 chrome                         C:\Program Files (x86)\Google\Chrome\Appli
  269.71875 ONENOTE                         C:\Program Files (x86)\Microsoft Office\Off
  188.59375 svchost                        C:\WINDOWS\System32\svchost.exe
  170.609375 powershell                     C:\Windows\System32\WindowsPowerShell\v1.0\p
  168.359375 lsass                          C:\WINDOWS\system32\lsass.exe
   148.5 svchost                         C:\WINDOWS\system32\svchost.exe
  143.890625 RuntimeBroker                 C:\Windows\System32\RuntimeBroker.exe
  140.65625 svchost                        C:\WINDOWS\System32\svchost.exe
  129.78125 chrome                         C:\Program Files (x86)\Google\Chrome\Appli

```

One of the features of *sort-object* is to provide a unique sort. In this case, *sort-object* will remove objects with duplicate sort keys. In this example, a csv file contains the information shown below. Ostensibly, this is a firewall log from we would like to determine the unique source IP address represented in the file.

Date	Time	Source_IP	Source_Port	Dest_IP	Dest_Port
Jan 16 2014	14:54:31	188.26.62.236	30870	207.75.134.154	8832
Jan 16 2014	14:54:31	193.110.18.33	62629	207.75.134.154	8832
Jan 16 2014	14:54:30	187.13.238.91	22350	207.75.134.154	8832
Jan 16 2014	14:54:29	46.249.171.169	27376	207.75.134.154	8832
Jan 16 2014	14:54:29	188.134.40.93	49611	207.75.134.154	8832
Jan 16 2014	14:54:29	187.13.238.91	13757	207.75.134.154	8832
Jan 16 2014	14:54:29	80.99.73.186	16008	207.75.134.154	8832
Jan 16 2014	14:54:29	187.13.238.91	48884	207.75.134.154	8832

In the script below, the *import-csv* cmdlet imports the csv file and creates custom objects with the properties corresponding to the columns in the csv file. The *sort-object* cmdlet sort the objects on the property *source_ip* then passes a unique set of objects, i.e., removes object with duplicate *source_ip* values, down the pipe to *select-object*. The cmdlet *select-object* the creates a new set of objects containing the *source_ip* property.

```
import-csv fwlog.csv |  
    sort-object -property source_ip -unique |  
    select-object source_ip
```

```
PS C:\Users>  
PS C:\Users> import-csv fwlog.csv |  
    >>> sort-object -property source_ip -unique | ... select-object sour  
  
Source_IP  
-----  
187.13.238.91  
188.134.40.93  
188.26.62.236  
193.110.18.33  
46.249.171.169  
80.99.73.186
```

In this next example, we find a unique set of file extensions in the path c:\windows and it's sub-directories,

```
get-childitem c:\windows -recurse -file -erroraction  
silentlycontinue |  
    sort-object -property extension -unique |  
    select-object extension | out-host -paging
```

```
PS C:\Users>
PS C:\Users> get-childitem c:\windows -recurse -file -erroraction silentlycor
>>>     sort-object -property extension -unique |
>>>     select-object extension | out-host -paging

Extension
-----
.0_none_f3d08e96a4fa8408_bootmgr_07e7e7fe
.0_none_f3d08e96a4fa8408_bootnxt_07e7ea74
.001
.002
.003
.1
.2
.2015-03-31-13-42-20-0100-00
.2015-04-23-14-58-19-0117-00
.2015-04-30-15-20-18-0505-00
.2015-05-27-13-05-00-0242-00
.2015-06-06-20-19-02-0562-00
.2015-07-08-18-39-58-0263-00
.2015-08-06-22-04-12-0160-00
.2015-08-27-17-09-39-0293-00
.3
.30319
.30319 64
.30319 64 Critical
.30319 critical
.3mf
.4
.7db
.7z
.acl
.acm
```

Sorting on one property is many times not sufficient. Sorting on multiple properties requires specifying the property names to the `-property` parameter as shown in the example below.

```
get-childitem c:\windows | sort-object -Property length, name -
descending
```

```

PS C:\Users>
PS C:\Users> get-childitem c:\windows | sort-object -Property Length, name -des
Directory: C:\windows

Mode                LastWriteTime         Length Name
----                -----          ----
-a---]   8/13/2015 4:54 AM      4544304 explorer.exe
-a---]   8/13/2015 4:54 AM      994816 HelpPane.exe
-a---]   8/13/2015 4:55 AM     316640 WMSysPr9.prx
-a---]   8/13/2015 4:55 AM     215040 notepad.exe
-a---]   8/13/2015 4:54 AM     156160 regedit.exe
-a---]   8/13/2015 4:54 AM    128000 sp1wow64.exe
                   9/15/2015 11:26 AM      75792 NgcPopKeySrv.log
-a--s-   9/16/2015 10:58 AM      67584 bootstat.dat
-a---]   8/13/2015 4:54 AM     61952 bfsvc.exe
-a---]   8/13/2015 4:55 AM     60416 twain_32.dll
-a----]   8/27/2015 5:55 PM     59396 comsetup.log
-a---]   8/13/2015 4:54 AM     43131 mib.bin
-a----]   8/13/2015 4:55 AM     32200 Enterprise.xml
-a----]   8/27/2015 5:58 PM     26673 diagwrn.xml
-a----]   8/27/2015 5:58 PM     26673 diagerr.xml
-a----]   9/2/2015 11:50 AM     19300 setupact.log
-a---]   8/13/2015 4:54 AM     18432 hh.exe
-a----]   5/27/2015 12:24 PM     17310 certenroll.log

```

The problem arises when we need to sort on two or more properties where each property is sorted in a different sequence. In this case, we need to use a has table to pass this information to the cmdlet. In the example below, we do a descending sort on the file extension and an ascending sort on the file name.

```

get-childitem c:\windows\system32 | select-object extension, name |
  sort-object @{e="extension";descending=$true},
@{e="name";ascending=$true} |
  format-table -autosize | out-host -paging

```

```

PS C:\Users>
PS C:\Users> get-childitem c:\windows\system32 | select-object extension, name
>>>     sort-object @{e="extension";descending=$true}, @{e="name";ascending=$true}
>>>     format-table -autosize | out-host -paging

Extension          Name
-----          -----
.xls             EventViewer_EventDetails.xls
.xls             WsmPty.xls
.xls             WsmTxt.xls
.xml              ApnDatabase.xml
.xml              AppxProvisioning.xml
.xml              NdfEventView.xml
.xml              NetTrace.PLA.Diagnostics.xml
.xml              OEMDefaultAssociations.xml
.xml              ScavengeSpace.xml
.xml              tcpbidi.xml
.xml              WdsUnattendTemplate.xml
.xml              wpr.config.xml
.xml              wsmanconfig_schema.xml
.xml              xpsrchvw.xml
.wsf              manage-bde.wsf
.vp               iglhxa64.vp
.vp               iglhxc64.vp
.vp               iglhxc64_dev.vp
.vp               iglhxg64.vp
.vp               iglhxg64_dev.vp
.vp               iglhxo64.vp
.vp               iglhxo64_dev.vp
.vp               iglhxs64.vp
.vbs              gatherNetworkInfo.vbs

```

Sorting a hash table requires some special handling. The example below demonstrates this clearly. Recall the hash tables are a key value pair where the property name for the key is name. To sort a hash table, we would be tempted to do the following.

```

$states = @{"Washington" = "Olympia"; "Oregon" = "Salem";
"California" = "Sacramento"}
$states | sort-object -property name

```

```

PS C:\Users>
PS C:\Users> $states = @{"Washington" = "Olympia"; "Oregon" = "Salem"; "California" = "Sacramento"}
PS C:\Users> $states | sort-object -property name

Name          Value
----          -----
Washington    Olympia
Oregon        Salem
California   Sacramento

```

At this point you may be wondering what happened. Let's alter the pipeline to get more information. We add *measure-object* to count the number of entries in the sorted hash table.

```
$states = @{"Washington" = "Olympia"; "Oregon" = "Salem";
"California" = "Sacramento"}
$states | sort-object -property name | measure-object
```

```
PS C:\Users>
PS C:\Users> $states = @{"Washington" = "Olympia"; "Oregon" = "Salem"; "California" = "Sacramento"
PS C:\Users> $states | sort-object -property name | measure-object

Count      : 1
Average    :
Sum        :
Maximum   :
Minimum   :
Property  :
```

As we see there is only one object exiting the pipeline because one object entered the pipeline. Collection objects, like arrays and hash tables, are passed down the pipe as one object. However, they have an enumerator method so that the collection is broken down into individual elements where are subsequently passed down the pipe.

```
$states.getenumerator() | sort-object -property name |
measure-object

$states.getenumerator() | sort-object -property name
```

```
PS C:\Users>
PS C:\Users> $states.GetEnumerator() | sort-object -property name |
>>>     measure-object

Count      : 3
Average    :
Sum        :
Maximum   :
Minimum   :
Property  :

PS C:\Users>
PS C:\Users> $states.GetEnumerator() | sort-object -property name

Name          Value
----          ---
California   Sacramento
Oregon       Salem
Washington   Olympia
```

Sometimes When sorting objects we would like to sort a unique set of objects, i.e., remove any duplicate objects. The *get-unique* cmdlet provides this capability. For this cmdlet to work properly, the input objects must be sorted on the property that on which *get-unique* operates.

The first example illustrates retrieving a unique set of process currently executing. First, let see the processes that are executing.

```
PS C:\Users>
PS C:\Users> get-process | out-host -paging
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	ID	ProcessName
498	21	4712	2028	...71	0.14	1984	ActionUriServer
481	19	4096	19504	...66	0.11	10932	ActionUriServer
337	16	4964	5956	95	0.14	10756	Amazon Music Helper
1389	126	153884	103652	...20	54.33	2872	ApplicationFrameHost
110	8	1164	1740	52		2740	armsvc
249	16	9732	11452	...00	...17.52	9788	audiodg
582	49	106072	37676	449	1.14	6132	Builder3D
264	19	15600	8076	92		4628	CaptureGenUSB
480	45	69880	92372	348	925.44	1528	chrome
228	38	158304	154876	351	22.53	1920	chrome
220	28	97720	86336	285	91.95	4152	chrome
219	88	561848	549300	741	...56.08	4176	chrome
213	21	29336	14888	201	2.50	4180	chrome
213	21	31396	20044	201	3.86	4184	chrome
213	21	25864	11396	198	1.70	4600	chrome
231	40	189328	176480	417	269.69	5356	chrome
279	35	76940	96520	296	3.56	5492	chrome
221	25	78700	74376	258	6.25	6240	chrome
215	22	33028	23712	204	5.97	6632	chrome
222	24	53692	44648	229	26.63	6788	chrome
232	26	53004	38356	238	3.64	6836	chrome
287	31	88144	63576	370	4.89	7592	chrome
213	21	26728	12788	198	2.13	7676	chrome
213	21	26800	12360	198	2.22	7908	chrome
220	22	30768	15964	211	2.80	8220	chrome
213	21	26928	12324	198	2.11	8224	chrome
213	22	31632	16272	212	2.35	8252	chrome

Notice the number of chrome processes that are active. Now observe the output of the pipeline below where the process objects are sorted and piped into *get-unique*.

```
get-process | sort-object name | get-unique
```

PS C:\Users> get-process sort-object name get-unique out-host -paging							
Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	ID	ProcessName
481	19	4096	19348	...66	0.11	10932	ActionUriServer
337	16	4964	5956	95	0.14	10756	Amazon Music Helper
1389	126	153884	103652	...20	54.33	2872	ApplicationFrameHost
110	8	1164	1740	52		2740	armsvc
249	19	9732	11452	...00	...20.02	9788	audiogd
582	49	106072	37676	449	1.14	6132	Builder3D
264	19	15600	8076	92		4628	CaptureGenUSB
214	22	33684	21892	210	4.20	8748	chrome
68	7	3584	8856	...06	2.52	7292	conhost
608	31	19748	33576	356	3.77	14748	CSISYN~1
786	29	2320	7688	...62		848	csrss
168	11	2344	2832	61		2764	CtHdaSvc
466	20	9412	15836	...24		2504	dasHost
191	12	5176	19844	...92	0.11	18712	DataExchangeHost
1091	85	93388	79592	...26		1188	dwm
2506	100	69904	130520	...35	374.38	5208	explorer
1035	42	18032	22064	208	...19.98	10220	g2mcomm
539	28	11832	11072	163	5.56	9952	g2mlauncher
327	18	4104	3652	130	0.05	10172	g2mstart
227	15	3912	7452	95		2796	GfExperienceService
273	20	5584	6228	161		2756	HauppaugeTVServer
306	21	3232	11448	112	0.11	9640	HPNetworkCommunicatorCom
277	18	11496	4780	155		2968	HPSupportSolutionsFrameworkS
166	11	1644	9132	82	0.05	11232	hpwusched2
732	46	34108	48380	425	0.63	19100	HxCalendarAppImm
798	54	48540	65900	500	0.75	9304	HxMail
739	30	11408	35036	182	29.23	13584	HxTsr
0	0	0	4	0		0	Idle
393	23	6228	3044	130	217.27	2568	ipoint
156							.

Note in the display above that only one chrome process is shown.

In this example, the csv file derive from a firewall log used in a previous section to demonstrate unique selection of objects is used here. We need to determine the unique source IP addresses represented in the file.

Date	Time	Syslog_ID	Source_IP	Source_Port	Dest_IP	Dest_Port
Jan 16 2014	14:54:31	106023	188.26.62.236		30870	207.75.134.154
Jan 16 2014	14:54:31	106023	193.110.18.33		62629	207.75.134.154
Jan 16 2014	14:54:30	106023	187.13.238.91		22350	207.75.134.154
Jan 16 2014	14:54:29	106023	46.249.171.169		27376	207.75.134.154

Jan 16 2014	14:54:29	106023	188.134.40.93		49611	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	187.13.238.91		13757	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	80.99.73.186		16008	207.75.134.154	8832
Jan 16 2014	14:54:29	106023	187.13.238.91		48884	207.75.134.154	8832

First, let's look at the incorrect implementation of the solution.

```
import-csv .\fwlog.csv | select-object source_ip |
    sort-object | get-unique
```

```
PS C:\Users>
PS C:\Users> import-csv .\fwlog.csv | select-object source_ip |
>>> sort-object | get-unique
Source_IP
-----
187.13.238.91
```

The output is obviously incorrect. The problem is that the csv file is piped as one large string into *get-unique*. We need to tell the cmdlet, with the *-asstring* parameter, that it is receiving a string and should look at individual lines in a string.

```
import-csv .\fwlog.csv | select-object source_ip |
    sort-object | get-unique -asstring
```

```
PS C:\Users>
PS C:\Users> import-csv .\fwlog.csv | select-object source_ip |
>>> sort-object | get-unique -asstring

Source_IP
-----
187.13.238.91
188.134.40.93
187.13.238.91
80.99.73.186
193.110.18.33
188.26.62.236
46.249.171.169
187.13.238.91
```

A final note on this cmdlet, both *sort-object* and *select-object* implement a *-unique* switch which accomplish the same functionality as the *get-unique* cmdlet. Using these switches eliminates the need for using the *get-unique* cmdlet.

Group

Monday, January 28, 2013
2:00 PM

Grouping is and has historically been one of the fundamental processes when working with data. The *group-object* cmdlet displays objects in groups by property value. Group allows aggregating data based on some property.

This first example shows a count of files by file extension. The second pipeline improves upon the first by excluding directories suppressing the group by using the *-noelement* switch parameter.

```
get-childitem c:\windows | group-object -property extension

get-childitem c:\windows -file | group-object -property extension -
noelement
```

```
PS C:\Users> get-childitem c:\windows | group-object -property extension

Count Name                                Group
---- 
76   .NET
1    .exe
9    .dat
10   .log
1    .reg
3    .xml
6    .ini
1    .bin
1    .dll
1    .prx

{addins, ADFS, appcompat, AppPatch...}
{Microsoft.NET}
{bfsvc.exe, explorer.exe, HelpPane.exe, hh.exe...}
{bootstat.dat}
{comsetup.log, DirectX.log, DtcInstall.log, DXError.log}
{CtHdaLoc.reg}
{diagerr.xml, diagwrn.xml, Enterprise.xml}
{game.ini, HCWPNP.INI, ODBC.INI, ODBCINST.INI...}
{mib.bin}
{twain_32.dll}
{WMSysPr9.prx}

PS C:\Users> get-childitem c:\windows -file | group-object -property extension -noe
```

In this example, we show the number of processes running by publisher. In the script, we use *select-object* to remove the group property emitted by the *group-object* cmdlet.

```
get-process | group-object -property company |  
    select-object count, name | format-table -autosize
```

```

PS C:\Users>
PS C:\Users> get-process | group-object -property company |
>>>     select-object count, name | format-table -autosize

Count Name
---- --
29 Microsoft Corporation
56
30 Google Inc.
3 Citrix Online, a division of Citrix Systems, Inc.
1 Hewlett-Packard
3
2 NVIDIA Corporation
1 Hewlett-Packard Development Company, LP
2 Valve Corporation
1 Hauppauge Computer Works, Inc.

```

In the next example, we want to categorize files by range of file size. This script shows the file size in megabytes and the number of files that are equal to or less than one megabyte smaller.

The first line in the pipeline gets the files objects then pipes then to sort. The objects are sorted in descending order so our output appears in the order of largest to smallest.

The second line creates a new property "LenCat", or length category, which normalizes the length into megabytes. Using the round function of the [math] to round file sizes to the closest whole megabyte. So, if a file size is 1.2 MB it falls into the category 1MB. If the size is 1.6, it's in the category 2MB. This is accomplished by the expression `$_.length / 1mb`.

The third line in the pipeline groups on the derived property lencat. Finally the output objects are piped to format-table for display. Since the *group-object* cmdlet outputs a name and count property, we use a hash table to change the column headings to something more descriptive.

```

get-childitem c:\windows\system32\*.dll | sort-object length -
descending |
    select-object @{name='LenCat'; expression={[math]::round($_.length
/ 1mb, 0) }} |

```

```
group-object -property lencat |
    format-table @{label='MB'; expression={$_['.name']} , @{label='# DLLs';e={$_['.count}}}
```

```
PS C:\Users>
PS C:\Users> get-childitem c:\windows\system32\*.dll | sort-object Length -descend
>>> select-object @{name='LenCat'; expression={[math]::round($_.Length / 1mb, 0)}}
>>> group-object -property lencat |
>>> format-table @{label='MB'; expression={$_['.name']} , @{label='# DLLs';e={$_['.co
MB # DLLs
-- -----
44      1
41      1
28      1
23      1
22      1
21      2
17      1
16      2
15      2
14      2
12      1
11      1
10      1
9       2
8       1
7       5
6       6
5      17
4      20
3      34
2      98
1     405
0     2356
```

The display is interpreted as one dll with a size between 43.5 and 44 MB, one dll with a size between 40.5 and 41 MB. Finally, there are 2,356 dlls less than 1MB in size.

The final example shows a grouping of files based on the number of days in the past that they were created.

In the pipeline, we group on the derived property "(get-date) - \$_.CreationTime).days" which calculates the number of days between the execution of the script and the creation time of the file. Since date arithmetic creates a time range object, we use a subexpression that we can select the days property. We use hash tables with *format-table* to provide more descriptive output. In these hash table note how column alignment and width may be specified.

```

get-childitem c:\windows\system32\*.dll -file |
    group-object -property { ((get-date) - $_.CreationTime).days } |
        format-Table @{{label="# of nfiles"; e={ $_.count}}
;align='right';width=7},
            @{{label="File age`n(days)"; e={ $_.name
}};align='right';width=7},
            @{{label="`nFiles"; e={ $_.group}};align='left';width=30} -autosize -wrap

```

```

PS C:\Users>
PS C:\Users> get-childitem c:\windows\system32\*.dll -file |
>>>     group-object -property { ((get-date) - $_.CreationTime).days } |
>>>     format-Table @{{label="# of nfiles"; e={ $_.count}};align='right';width=7},
>>>             @{{label="File age`n(days)"; e={ $_.name}};align='right';width=7},
>>>             @{{label="`nFiles"; e={ $_.group}};align='left';width=30} -autosize -wrap

```

# of files	File age (days)	Files
2513	68	{C:\windows\system32\aaauthhelper.dll, C:\windows\system32\aaadb.dll, C:\windows\system32\AboveLoc...
5	132	{C:\windows\system32\accesor.dll, C:\windows\system32\ncs2instutility.dll, C:\windows\system3...
15	8	{C:\windows\system32\acmigration.dll, C:\windows\system32\authui.dll...}
167	33	{C:\windows\system32\ACPBackgroundManagerPolicy.dll, C:\windows\system32\ActiveSyncProvider.dll, C:\windows\system3...
9	755	{C:\windows\system32\api-ms-win-core-fibers-12-1-1.dll, C:\windows\system32\api-ms-win-core-psm-appnotify-11-1-0.dll, C:\windows\system32\api-ms-win-devices-config-11-1-1.dll, C:\windows\system32\api-ms-win-mm-misc-11-1-1.dll...}
41	29	{C:\windows\system32\AppXDeploymentClient.dll, C:\windows\system32\CloudAP.dll, C:\windows\system32\diagtr...
26	19	{C:\windows\system32\AppXDeploymentExtensions.dll, C:\windows\system32\BthRadioMedia.dll, C:\windows\system32\...
18	1559	{C:\windows\system32\at1100.dll, C:\windows\system32\mfcc100cht.dll...}
5	69	{C:\windows\system32\cmipnpinstall.dll, C:\windows\system32\cmipnpnp.dll, C:\windows\system32\SSShim.dll, C:\windows\system32\sxs.dll}
2	617	{C:\windows\system32\coin95ip.dll, C:\windows\system32\coin97ip.dll, C:\windows\system32\coin...
2	119	{C:\windows\system32\coin97ip.dll, C:\windows\system32\coin...
3	89	{C:\windows\system32\ctdco64.dll, C:\windows\system32\cthd...
29	208	{C:\windows\system32\d3dcompiler_33.dll, C:\windows\system32\d3dx10.dll, C:\windows\system32\d3dx10...
59	201	{C:\windows\system32\d3dcompiler_35.dll, C:\windows\system32\d3dcompiler_37.dll, C:\windows\system32\d3dcompiler_3...

Measure

Monday, January 28, 2013

9:34 PM

The *measure-object* cmdlet provides basic metrics for a group of objects. The metrics are count, sum, average, max, and min.

The first example provides metrics for the files in the designated directory.

```
get-childitem c:\windows -file | measure-object length -average -sum  
-max
```

```
PS C:\Users>  
PS C:\Users> get-childitem c:\windows -file | measure-object Length -average -sum  
  
Count      : 33  
Average    : 269672.909090909  
Sum        : 8899206  
Maximum    : 4532304  
Minimum    :  
Property   : Length
```

We interpret the above as 33 files are present in the c:\windows directory. The average size of the files is about 27KB. The total size of the files is about 89MB. The largest file is about 4.5MB. The smallest file has zero size.

This example calculates the average and maximum virtual memory of all running processes.

```
get-process | measure-object virtualmemorysize64 -average -max |  
    format-list @{l='Average VM';e={$_.average}}, @{l='Maximum  
VM';e={$_.maximum}}
```

```
PS C:\Users>  
PS C:\Users> get-process | measure-object virtualmemorysize64 -average -max  
>>>     format-list @{l='Average VM';e={$_.average}}, @{l='Maximum VM';e={$_.ma  
  
Average VM : 850736810108.121  
Maximum VM : 2235225980928
```

The *measure-object* cmdlet is also very useful for working with text files. This cmdlet can measure the number of characters, words, and lines in a file. In this final example, we produce metrics for a log file.

```
get-content c:\windows\windowsupdate.log |  
measure-object -line -character -word
```

```
PS C:\Users>  
PS C:\Users> get-content c:\windows\windowsupdate.log |  
>>> measure-object -line -character -word  


| Lines | Words | Characters | Property |
|-------|-------|------------|----------|
| 3     | 32    | 267        |          |


```

Compare

Friday, July 18, 2014
11:00 AM

The *compare-object* cmdlet is used to compare objects and display the differences. This is very useful as shown in the examples below.

This first example illustrates a scenario where we want to know that processes that started during the last measurement interval. To do this, we need to capture the process objects that are executing at the beginning of the interval then capture the running process at the end of the interval. We then use *compare-object* to see the differences.

```
$before = get-process  
start-sleep -seconds 30  
$after = get-process  
compare-object -ReferenceObject $before -DifferenceObject $after |  
format-table -autosize
```

```

PS C:\Users> $before = get-process
PS C:\Users> start-sleep -seconds 30
PS C:\Users> $after = get-process
PS C:\Users> compare-object -ReferenceObject $before -DifferenceObject $after | 
>>>     format-table -autosize

InputObject                               SideIndicator
-----
System.Diagnostics.Process (mspaint) =>
System.Diagnostics.Process (notepad) =>
System.Diagnostics.Process (WINWORD) =>

```

The *Side Indicator* indicates the location of the differences. The => indicates that the differences are in the *DifferenceObject*, the variable \$after. We interpret this to mean that the processes mspaint, notepad, and WINWORD are executing now but weren't executing before.

In the next example, we compare two files. Given file1.txt and file2.txt exist with the contents shown below.

<u>File1</u>	<u>File2</u>
Line 01	Line 01
Line 02	Line 03
Line 04	Line 04
Line 05	Line 06
Line 07	Line 07

The following script compares the two files. We use the -IncludeEqual switch to show the lines that are present in both files. We use subexpression to retrieve the contents of both files. We could have chosen to store the contents of each file and separate variables. Then used those variables as parameters to *compare-object*.

```

compare-object -ReferenceObject (get-content file1.txt) ` 
    -DifferenceObject (get-content file2.txt) -IncludeEqual | format-
table -autosize

```

```

PS C:\Users>
PS C:\Users> compare-object -ReferenceObject (get-content file1.txt) ` 
>>>      -DifferenceObject (get-content file2.txt) -IncludeEqual |format-table -auto
InputObject SideIndicator
-----
Line 01    ==
Line 04    ==
Line 07    ==
Line 03    =>
Line 06    =>
Line 02    <=
Line 05    <=

```

The output is interpreted as follows. The == means that the line is in both files. The => means that the line is in file2.txt but not in file1.txt. The <= means that the line is in file1.txt but not in file2.txt.

Note: before using the get-content cmdlet always be aware of the size of the text file. The get-cmdlet reads the complete contents of the file into memory. If the file is large, the cmdlet may take sometimes to complete and may cause significant virtual memory paging.

The final example is very useful since it allows us to compare the contents to two directories and determine the differences. In this example, we use variables to store the files system objects before comparing them. We could have chosen to use subexpressions as in the previous example.

```

compare-object -ReferenceObject (get-childitem d:\mytemp\dir1) ` 
-DifferenceObject (get-childitem d:\mytemp\dir2) -IncludeEqual
|format-table -autosize

```

```

PS C:\Users>
PS C:\Users> compare-object -ReferenceObject (get-childitem d:\mytemp\dir1) ` 
>>>      -DifferenceObject (get-childitem d:\mytemp\dir2) -IncludeEqual |format-table -a
InputObject      SideIndicator
-----          -----
cis100.docx     ==
Login_to_GApps.pdf =>
Screenshot.png   =>
instructions.docx  <=

```

The output is interpreted as follows. The file cis100.docx is present in both directories. The files Login_to_Gapps.pdf and Screenshot.png are present in dir2. The file instructions.docx is present in dir1.

Existence Test

Friday, August 08, 2014
3:45 PM

The *test-path* cmdlet may be used to test for the existence of objects on an PowerShell drive. This cmdlet returns \$true or the referenced object exists or false otherwise.

```
test-path d:\mytemp\dir2\screenshot.png
test-path d:\mytemp\dir1\screenshot.png
test-path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
test-path d:\mytemp\dir2\screenshot.png -newerthan '08-14-2015'
test-path d:\mytemp\dir2\screenshot.png -olderthan '08-14-2015'
```

```
PS C:\Users>
PS C:\Users> test-path d:\mytemp\dir2\screenshot.png
True
PS C:\Users>
PS C:\Users> test-path d:\mytemp\dir1\screenshot.png
False
PS C:\Users>
PS C:\Users> test-path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
True
PS C:\Users>
PS C:\Users> test-path d:\mytemp\dir2\screenshot.png -newerthan '08-14-2015'
True
PS C:\Users>
PS C:\Users> test-path d:\mytemp\dir2\screenshot.png -olderthan '08-14-2015'
False
PS C:\Users>
```

The value of this cmdlet is seen in later sections where we learn to develop scripts.

Converting Results

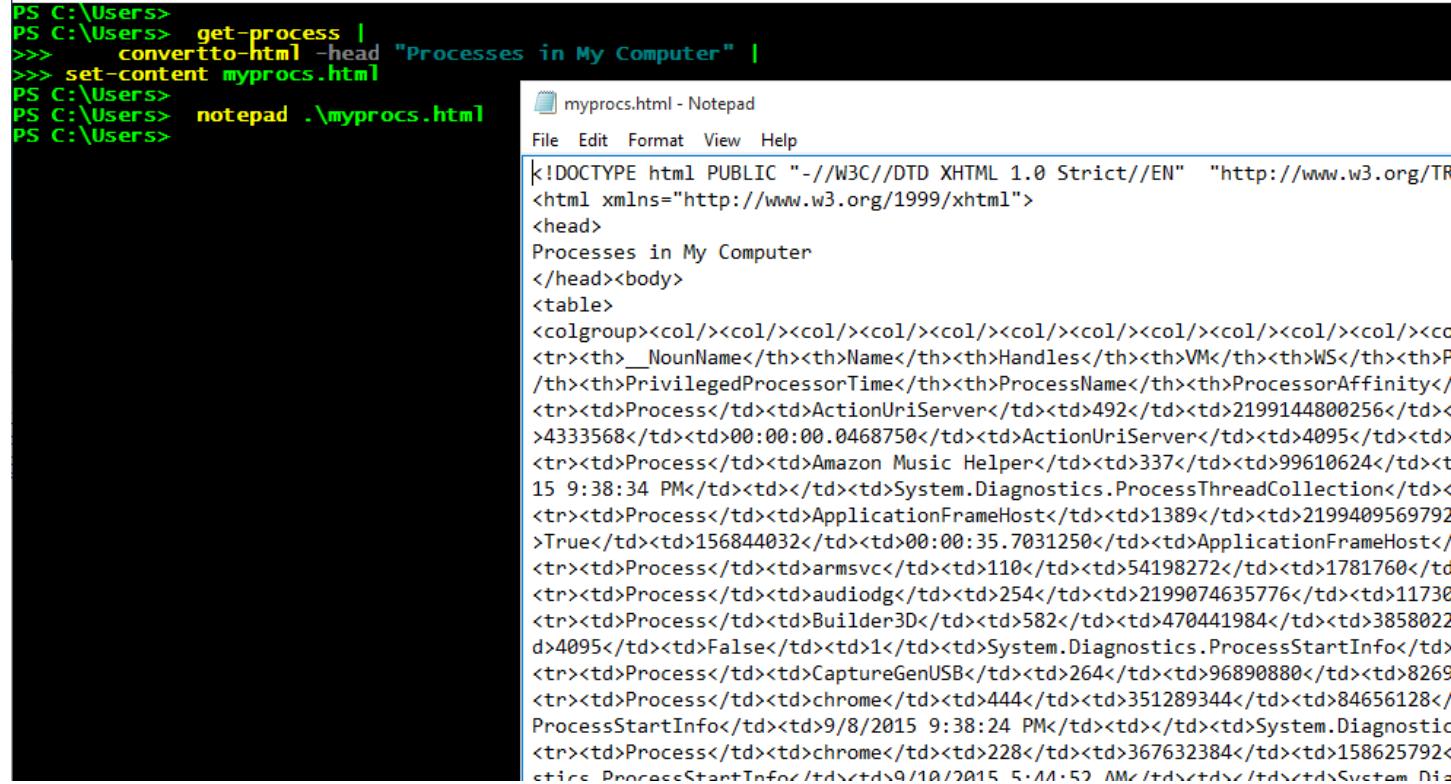
Tuesday, January 29, 2013
7:41 AM

There are a number of cmdlets that convert objects. The more useful ones are listed below.

1. `convertto-html` Convert output objects to html
2. `convertto-csv` Convert output objects to a comma delimited file
3. `convertfrom-csv` Convert csv strings to PowerShell objects

This example converts the output of the `get-process` cmdlet to an html file. We add a page heading with the `-head` parameter.

```
get-process |  
    convertto-html -head "Processes in My Computer" |  
    set-content myprocs.html
```



```
PS C:\Users>  
PS C:\Users> get-process |  
->>> convertto-html -head "Processes in My Computer" |  
->>> set-content myprocs.html  
PS C:\Users>  
PS C:\Users> notepad .\myprocs.html  
PS C:\Users>
```

The screenshot shows a Windows command prompt window with the following session:

```
PS C:\Users>  
PS C:\Users> get-process |  
->>> convertto-html -head "Processes in My Computer" |  
->>> set-content myprocs.html  
PS C:\Users>  
PS C:\Users> notepad .\myprocs.html  
PS C:\Users>
```

To the right of the command prompt, a Notepad window is open with the title "myprocs.html - Notepad". The content of the file is an HTML document with a page header and a table containing process information. The table has columns for ProcessName, Handles, VM, WS, ProcessorAffinity, and other details. The Notepad window also shows the file's properties and a preview of the document.

The resultant page when viewed in a browser appears as below.

Processes in My Computer							
	Name	Handles	VM	WS	PM	NPM	Path
Process	ActionUriServer	492	2199144800256	1736704	4333568	19456	C:\Windows\SystemApps\Microsoft.Windows.Cortana_cw5n1h2txyewy\ActionU
Process	Amazon Music Helper	337	99610624	6098944	5083136	16760	C:\Users\mike\AppData\Local\Amazon Music\Amazon Music Helper.exe
Process	ApplicationFrameHost	1389	2199409569792	106393600	156844032	129056	C:\WINDOWS\system32\ApplicationFrameHost.exe
Process	armsvc	110	54198272	1781760	1191936	8080	
Process	audiogd	254	2199074635776	11730944	9842688	16344	
Process	Builder3D	582	470441984	38580224	108617728	49888	C:\Program Files\WindowsApps\Microsoft.3DBuilder_10.9.6.0_x64_8wekyb3d
Process	CaptureGenUSB	264	96890880	8269824	15974400	19088	
Process	chrome	444	351289344	84656128	70246400	40760	C:\Program Files (x86)\Google\Chrome\Application\chrome.exe
Process	chrome	228	367632384	158625792	162234368	38480	C:\Program Files (x86)\Google\Chrome\Application\chrome.exe
Process	chrome	220	300064768	89042944	100384768	28960	C:\Program Files (x86)\Google\Chrome\Application\chrome.exe

This example illustrates the use of the `convertto-csv` cmdlet.

```
get-process |  
    convertto-csv |  
    set-content myprocs.csv
```

```

PS C:\Users>
PS C:\Users> get-process |
>>>     convertto-csv |
>>> set-content myprocs.csv
PS C:\Users>
PS C:\Users> Invoke-item myprocs.csv
PS C:\Users>

```

The screenshot shows a Microsoft Excel spreadsheet titled "myprocs.csv - Excel". The data consists of two rows of headers and eleven rows of process information. The columns are labeled A through J. The first column contains the process ID (1-11), the second column contains the process name (e.g., Process, chrome), and the third column contains the handle count (e.g., 492, 228). Subsequent columns provide various performance metrics like VM, WS, PM, NPM, Path, Company, CPU, and File.

	#TYPE System.Diagnostics.Process									
1	#TYPE System.Diagnostics.Process									
2	_NounName	Handles	VM	WS	PM	NPM	Path	Company	CPU	File
3	Process	ActionUri	492	2.2E+12	1503232	4333568	19456	C:\Wind	Microsoft	0.109375 10.0
4	Process	Amazon N	337	99610624	6098944	5083136	16760	C:\Users\mike\AppD		0.15625
5	Process	Applicatio	1418	2.2E+12	1.08E+08	1.6E+08	130960	C:\WINDC	Microsoft	55.01563 10.0
6	Process	armsvc	110	54198272	1781760	1191936	8080			
7	Process	audiogd	248	2.2E+12	11669504	9809920	15800			10729.28
8	Process	Builder3D	582	4.7E+08	38576128	1.09E+08	49888	C:\Progra	Microsoft	1.140625 10.9
9	Process	CaptureGe	264	96890880	8269824	15974400	19088			
10	Process	chrome	444	3.51E+08	84656128	70246400	40760	C:\Progra	Google In	926.0156 45.0
11	Process	chrome	228	3.68E+08	1.59E+08	1.62E+08	38480	C:\Progra	Google In	22.64063 45.0

The next example demonstrates the use of the convertfrom-csv cmdlet. The cmdlet creates a custom object using a string in csv format.

In the script below the pipeline generates a string for each process object. The string consists of the process name, a comma, the CPU time for the process, and a newline character. For the sake of expediency, only the first five process objects are selected. After all objects in the pipeline are consumed, \$s is a string consisting of the five substrings described above. After the string is created, a string that contains the label "name", a comma, and the label "CPU" is prepended to the string created in the previous pipeline. This string is equivalent to the contents of a csv file as show in the first part of the display output. The string \$s is then converted to a custom object using the convertfrom-csv cmdlet. The custom object contains the usual inherited methods plus the noteproperties name, and CPU.

```

clear-host
write-host `n`n
[string] $s = ""
$s = get-process | where { $_.cpu -gt 0} |
    select-object -first 5 | foreach { "$($_.name),"
$_.cpu`n" }
$s = "Name, CPU`n" + $s
$s

```

```

1 clear-host
2 write-host `n`n
3 [string] $s = ""
4 $s = get-process | where { $_.cpu -gt 0} |
5     select-object -first 5 | foreach { "$($_.name), $($_.cpu`n" )
6 $s += "Name, CPU`n" + $s
7 $s
8 $c = convertfrom-csv $s
9 $c
10

Name, CPU
ApplicationFrameHost, 66.1875
AppVShNotify, 0.140625
audiogd, 8751.3125
browser_broker, 1.671875
chrome, 6

Name          CPU
----          ---
ApplicationFrameHost 66.1875
AppVShNotify      0.140625
audiogd           8751.3125
browser_broker     1.671875
chrome            6

```

Drive Abstraction

Sunday, August 4, 2013
11:42 AM

The Windows OS supports a number of hierarchical data stores, the most common being the file systems such as FAT, in its various incarnations, and NTFS. These file systems are navigated and managed using the familiar cd and md commands in the cmd shell.

However, there are a number of stores such as the registry, Active Directory, and others that are also hierarchical. The cmd shell provides nothing that allows the user to navigate or manage these databases.

PowerShell abstracts many of these databases so that they are accessible using the same cmdlet regardless of type of database. The abstracted drives are called PowerShell drives or PSDrives and are made available by the PowerShell Providers called PSProviders.

PowerShell also abstracts repositories that are useful when writing scripts such as:

Alias: While this may seem odd, this virtual drive is useful and an easy way to discover the aliases associated with cmdlets.

Cert: The repository of all certificates on this host.

Env: The container for all environment variables. Having environment variables available through this repository facilitates the dynamic creation and manipulation of these variables within a script.

<http://technet.microsoft.com/en-us/library/dd315335.aspx>

The cmdlet **get-psdrive** retrieves the properties of the available abstracted drives. The illustration below shows some of the abstracted drives available. All of the drives listed below are navigated and managed using the same cmdlets. Shown below is a drive called tmp:. This drive points to the directory MyTemp in the root of volume D. This illustrates another powerful feature of PowerShell where virtual drives may be created that point to a location of any of the data stores supported by the PowerShell Providers.

Name	Used (GB)	Free (GB)	Provider	Root
Alias			Alias	
B	669.48	262.03	FileSystem	B:\
C	168.25	308.17	FileSystem	C:\
Cert			Certificate	\
D	48.99	90.75	FileSystem	D:\
E	3.83		FileSystem	E:\
Env			Environment	
F			FileSystem	F:\
Function			Function	
G			FileSystem	G:\
H			FileSystem	H:\
HKCU			Registry	HKEY_CURRENT_USER
HKLM			Registry	HKEY_LOCAL_MACHINE
I	903.15	959.87	FileSystem	I:\
J			FileSystem	J:\
K			FileSystem	K:\
L			FileSystem	L:\
N	204.78	74.56	FileSystem	N:\
tmp		90.75	FileSystem	D:\MyTemp
Variable			Variable	
WSMan			WSMan	

The parameters of the cmdlet provide the ability to filter what is displayed:

-literalname The specified string is used to match against drive names. No wildcards

```
gdr -literalname c
```

-name Name string may contain wildcards. This is a positional parameter

```
gdr -name 'cert*'
```

-PSProvider Get drives only for the specified provider

```
gdr -psprovider filesystem
```

```
gdr -psprovider registry
```

We can use Get-PSdrive to list other properties of the desired drive. In the example below, the properties of drive "c" are queried and piped to the format-table cmdlet to provide a tabular display of the selected properties.

```
get-psdrive c | format-table name, free -autosize
```

```
PS C:\Users>
PS C:\Users> get-psdrive c | format-table name, free -autosize
Name          Free
----          ----
C      241745059840
```

Let's change the free space from bytes to gigabytes. This is done by dividing the free (free space) property by 1GB. To do that we need to use a hash table as shown below.

```
get-psdrive c | format-table name,
@{Label="Free Space(GB)";Expression={$_.free / 1gb}} -autosize
```

```
PS C:\Users>
PS C:\Users> get-psdrive c | format-table name,
@{Label="Free Space(GB)";Expression={$_.free / 1gb}} -autosize
>>
>>
Name  Free Space(GB)
----  -----
C    225.141422271729
```

In the above script, the drive object representing the volume c is piped to format-table. Format-Table outputs information in table format. The first column displayed is the value of the *name* property. The syntax after the "," and on the next line is a called a hash-table and allows a *derived property* to be created. The

Label identifies the column heading, "Free Space(GB)". The Expression is the value to display in this column. In the case, we take the free property of the current object in the pipeline (\$_.free) and divide that 1GB. In the assignment statement for Expression, the arithmetic expression (\$_.free / 1gb) is delimited by an opening and a closing brace. This is called a *script block*. As the name implies we may have more complex PowerShell script that returns one value to assign to Expression.

<http://technet.microsoft.com/en-us/library/dd347677.aspx>

Note, also in the above image that, since the first line terminated by a comma, PowerShell knows more input is required so the continuation prompt >> was displayed.

The format-list cmdlet is used to display a columnar list of properties. In the first execution below, only the default properties are displayed in the console. In the second execution, the wildcard * is used causing format-list to display all properties. A comma separated list of property names may be specified with format-list in which case only those properties are listed.

```

PS C:\Users>
PS C:\Users> gdr c | format-list

Name          : C
Description   : OS
Provider      : Microsoft.PowerShell.Core\FileSystem
Root          : C:\
CurrentLocation : Users


PS C:\Users> gdr c | fl *

Used          : 257536860160
Free          : 241741549568
CurrentLocation : Users
Name          : C
Provider      : Microsoft.PowerShell.Core\FileSystem
Root          : C:\
Description   : OS
Credential    : System.Management.Automation.PSCredential
DisplayRoot   :

```

Many of the properties of a PSDrive are read/write so they may be changed. The CurrentLocation property shows the current attach point on that drive. Notice that the script was executed in the directory c:\users. The CurrentLocation value is "users". This property changes as we navigate the directory tree.

Let's look at how to use the properties to change alter the state of the drive object.

```

d:                      # the prompt shows that we are at the root
c:                      # navigate back to c:
$d = get-psdrive d       # get the drive info object and assign it
                         # to $d
$d.CurrentLocation       # display the current location on the
                         # console (root)
$d.CurrentLocation =    # Change the current location to MyTemp,
'MyTemp'                 # i.e., d:\MyTemp
$d.CurrentLocation       # the property value is no 'MyTemp'

```

```
d: # navigate back to d and notice the prompt  
has change
```

```
PS C:\Users> d:  
PS D:\>  
PS D:\> c:  
PS C:\Users> $d = get-psdrive d  
PS C:\Users> $d | get-member  
  
TypeName: System.Management.Automation.PSDriveInfo  


| Name            | MemberType     | Definition                                                         |
|-----------------|----------------|--------------------------------------------------------------------|
| CompareTo       | Method         | int CompareTo(System.Management.Automation.PSDriveInfo drive), int |
| Equals          | Method         | bool Equals(System.Management.Automation.PSDriveInfo drive), bool  |
| GetHashCode     | Method         | int GetHashCode()                                                  |
| GetType         | Method         | type GetType()                                                     |
| ToString        | Method         | string ToString()                                                  |
| Credential      | Property       | pscredential Credential {get;}                                     |
| CurrentLocation | Property       | string CurrentLocation {get;set;}                                  |
| Description     | Property       | string Description {get;set;}                                      |
| DisplayRoot     | Property       | string DisplayRoot {get;}                                          |
| Name            | Property       | string Name {get;}                                                 |
| Provider        | Property       | System.Management.Automation.ProviderInfo Provider {get;}          |
| Root            | Property       | string Root {get;}                                                 |
| Free            | ScriptProperty | System.Object Free {get=## Ensure that this is a FileSystem drive} |
| Used            | ScriptProperty | System.Object Used {get=## Ensure that this is a FileSystem drive} |

  
PS C:\Users> $d.CurrentLocation  
PS C:\Users> $d.CurrentLocation = 'MyTemp'  
PS C:\Users> $d.CurrentLocation  
MyTemp  
PS C:\Users> PS C:\Users> d:  
PS D:\MyTemp>
```

We may shortcut these steps using a PowerShell sub-expression:

```
PS C:\Users>  
PS C:\Users> d:  
PS D:\>  
PS D:\> c:  
PS C:\Users>  
PS C:\Users> (gdr d).CurrentLocation = 'MyTemp'  
PS C:\Users>  
PS C:\Users> d:  
PS D:\MyTemp>
```

PowerShell interprets the above as follows:

gdr d

Create the PSDriveInfo object for c:

(gdr d)	By enclosing the cmdlet in parentheses, we create a PowerShell sub-expression. This causes PowerShell to execute the commands delimited by the parentheses. The objects that result from the execution of the sub-expression are saved in a temporary storage in the address space.
(gdr d).CurrentLocation = ...	This assigns this string 'MyTemp' to the CurrentLocation property of the saved temporary object (variable). Thereby causing the current location on the file system to change to the directory specified by the string 'd:\MyTemp'.

Navigation:

As we saw in the above examples, navigation between drives on a file system is executed in the same manner as the DOS or cmd shell. However, this only works on file system drives. To navigate to a drive other than the file system drive requires using the set-location cmdlet. The cmdlet **set-location** and its alias **cd** or **sl** allows the user to navigate to a PSDrive and to *walk the tree*.

<http://technet.microsoft.com/en-us/library/ee176962.aspx>

The example below illustrates navigate the registry hive HKEY_LOCAL_MACHINE

```
PS C:\Users>
PS C:\Users> set-location hklm:
PS HKLM:>
PS HKLM:> sl software
PS HKLM:\software>
PS HKLM:\software> cd microsoft
PS HKLM:\software\microsoft>
PS HKLM:\software\microsoft> set-location windows
PS HKLM:\software\microsoft\windows>
PS HKLM:\software\microsoft\windows> sl currentversion
PS HKLM:\software\microsoft\windows\currentversion>
PS HKLM:\software\microsoft\windows\currentversion> gci
```

Hive: HKEY_LOCAL_MACHINE\software\microsoft\windows\currentversion

Name	Property
AccountPicture	AppsReadAccess : 1
AdvertisingInfo	Enabled : 1
App Management	
App Paths	EnableWebContentEvaluation : 1
AppHost	
Applets	
AppModel	Version : 0
AppReadiness	NotifyObject : {9c212ed3-cfd2-4676-92d8-3fbb2c3a8379}
Appx	PackageRepositoryRoot : C:\ProgramData\Microsoft\Win
Audio	ExtensionsModule : C:\Windows\System32\AppXDeplo
	PackageRoot : C:\Program Files\Windows Apps
	IUILayoutPolicyModule : C:\Windows\System32\IUILP.dl
	VolumeRepeatWindow : 300
	EnableCaptureMonitor : 1
	VolumeDownTransitionTime : 2000
	VolumeUpTransitionTime : 0
	VolumeAccelThreshold : 6

We may also navigate to the same registry key using a fully qualified name. Since typing a fully-qualified name would be very tedious, we may use autocomplete to expedite the entry. Autocomplete is done entering part of the key then pressing the tab:

```
gci hklm:\soft<tab>\mic<tab>\win<tab>\cur<tab>
```

```
PS C:\Users>
PS C:\Users> set-Location HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion
PS HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion> gci

    Hive: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion

Name                           Property
----                           -----
AccountPicture
AdvertisingInfo
App Management
App Paths
AppHost
Applets
AppModel
AppReadiness
Appx

Property
-----
AppsReadAccess : 1
Enabled : 1

EnableWebContentEvaluation : 1

Version : 0
NotifyObject : {9c212ed3-cfd2-4676-92d8-3fbb2
PackageRepositoryRoot : C:\ProgramData\Micros
ExtensionsModule      : C:\Windows\System32\A
PackageRoot           : C:\Program Files\Wind
IUILayoutPolicyModule : C:\Windows\System32\I
```

The next example shows navigation of the certificates store. The certificates store holds all of the certificates for all users and for the computer. Certificates are used in securing communications between a user and host or between the local machine and another host.

In the illustration below, we navigate to the CurrentUser store and list the contents.

```
PS C:\Users>
PS C:\Users> set-location cert:
PS Cert:>
PS Cert:>\ gci

Location : CurrentUser
StoreNames : {TrustedPublisher, ClientAuthIssuer, Root, MSIEHistoryJournal}

Location : LocalMachine
StoreNames : {TrustedPublisher, ClientAuthIssuer, Root, TrustedDeviceOwner}

PS Cert:>\ set-location currentUser
PS Cert:\currentUser>
PS Cert:\currentUser>\ gci

Name : TrustedPublisher
Name : ClientAuthIssuer
Name : Root
Name : MSIEHistoryJournal
Name : CA
Name : UserDS
Name : AuthRoot
Name : TrustedPeople
Name : ADDRESSBOOK
Name : My
Name : SmartCardRoot
Name : Trust
Name : Disallowed
```

The AuthRoot store holds the certificates of the trusted root Certificate Authorities. Navigating down to the node and displaying the contents with get-childitem lists the certificates stored there. We pipe the result of get-childitem into the cmdlet sort-object to sort the certificates on the property subject.

```
PS Cert:\CurrentUser>
PS Cert:\CurrentUser> sl .\AuthRoot
PS Cert:\CurrentUser\AuthRoot>
PS Cert:\CurrentUser\AuthRoot> gci | sort subject
```

Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\AuthRoot

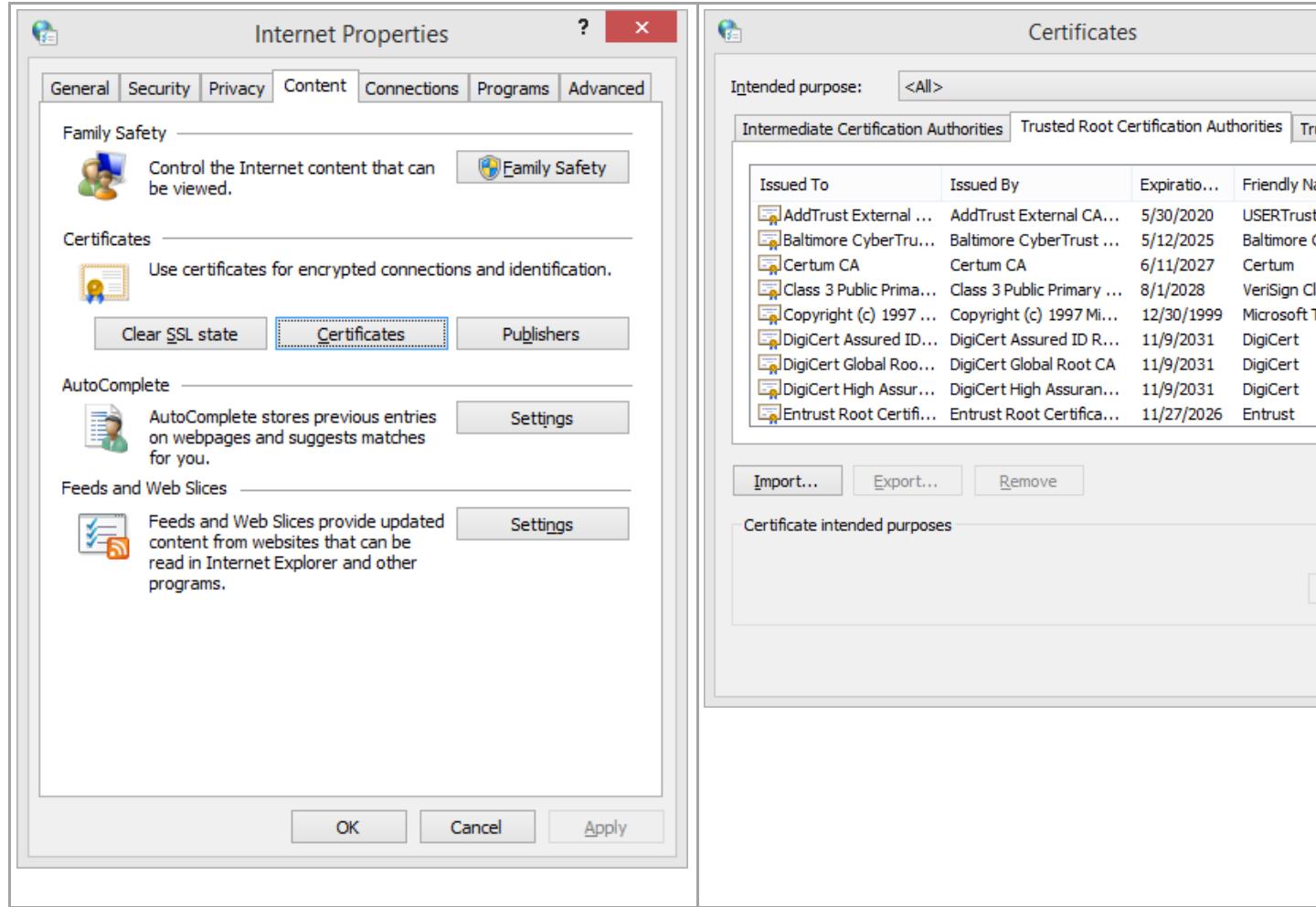
Thumbprint

```
-----  
02FAF3E291435468607857694DF5E45B68851868  
D4DE20D05E66FC53FE1A50882C78DB2852CAE474  
6252DC40F71143A22FDE9EF7348E064251B18118  
0563B8630D62D75ABBC8AB1E4BDFB5A899B24D43  
A8985D3A65E5E5C4B2D7D66D40C6DD2FB19C5436  
5FB7EE0633E259DBAD0C4C9AE6D38F1A61C7DC25  
B31EB1B740E36C8402DADC37D44DF5D4674952F9  
503006091D97D4F5AE39F7CBE7927D7D652D3431  
99A69BE61AFE886B4D2B82007CB854FC317E1539  
DE28F4A4FFE5B92FA3C503D1A349A7F9962A8212  
039EEDB80BE7A03C6953893B20D2D9323A4C2AFD  
B1BC968BD4F49D622AA89A81F2150152A41D829C  
D69B561148F01C77C54578C10926DF5B856976AD  
47BEABC922EAE80E78783462A79F45C254FDE68B  
07817050D81605706C24D800CE794421267EE474
```

Subject

```
-----  
CN=AddTrust External CA Root, OU=AddTrust External CA Root  
CN=Baltimore CyberTrust Root, OU=CyberTrust Root  
CN=Certum CA, O=Unizeto Sp. z o.o., O=Unizeto Group  
CN=DigiCert Assured ID Root CA, OU=digital certificate  
CN=DigiCert Global Root CA, OU=www.digicert.com  
CN=DigiCert High Assurance EV Root CA  
CN=Entrust Root Certification Authority  
CN=Entrust.net Certification Authority  
CN=Entrust.net Secure Server Certification Authority  
CN=GeoTrust Global CA, O=GeoTrust Inc.  
CN=GeoTrust Primary Certification Authority  
CN=GlobalSign Root CA, OU=Root CA, O=GlobalSign  
CN=GlobalSign, O=GlobalSign, OU=GlobalSign  
CN=Go Daddy Root Certificate Authority  
CN=GTI CyberTrust Global Root, OU="GTI CyberTrust Global Root"
```

For comparison purposes, the traditional method to view the trusted root CAs is to use either Internet Explorer or Internet Options in the Control Panel. In IE, navigate to Tools -> Internet Options -> Content then click the Certificates button. Select the Trusted Root Certification Authorities tab.



Creating a PSDrive:

The cmd shell provides the `subst` command to create a virtual drive pointing to a directory on the local file system. The drive name was restricted to one letter to conform to the standard Windows drive naming convention. PowerShell does away with this convention. Using the `New-Psdrive` cmdlet, the user may create a new virtual drive with a descriptive name pointing to a directory on the either local or remote host, the registry, the certificate store, or Active Directory.

```
New-PsDrive -name mytemp -PSProvider FileSystem -root 'd:\mytemp'
```

```

PS C:\Users>
PS C:\Users> New-PSDrive -name mytemp -PSProvider FileSystem -root 'd:\mytemp'

```

Name	Used (GB)	Free (GB)	Provider	Root
mytemp		90.72	FileSystem	D:\mytemp

Note the use of the back tick in the command below.

```

New-PSDrive -name HKLM_Run -PSProvider Registry ` 
    -root 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion'

```

```

PS C:\Users>
PS C:\Users> New-PSDrive -name HKLM_Run -PSProvider Registry ` 
>> -root 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion'
>>


```

Name	Used (GB)	Free (GB)	Provider	Root
HKLM_Run			Registry	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion

To create a PS drive pointing to a remote host, use a UNC name in the root. This is the equivalent of the drive mapping and is subject to credential checking just like a drive map.

```

New-PSDrive -name LTop -PSProvider FileSystem -root
'\\IS017871\users'

```

```

PS C:\Users>
PS C:\Users> New-PSDrive -name LTop -PSProvider FileSystem -Root '\\IS017871\users'

```

Name	Used (GB)	Free (GB)	Provider	Root
LTop			FileSystem	\\IS017871\users

Using Get-PSDrive to list the drives we see that the new drives are available.

Name	Used (GB)	Free (GB)	Provider	Root
Alias			Alias	
B	671.34	260.17	FileSystem	B:\
C	165.45	310.97	FileSystem	C:\
Cert			Certificate	\
D	49.02	90.72	FileSystem	D:\
E	3.83		FileSystem	E:\
Env			Environment	
F			FileSystem	F:\
Function			Function	
G			FileSystem	G:\
H			FileSystem	H:\
HKCU			Registry	HKEY_CURRENT_USER
HKLM			Registry	HKEY_LOCAL_MACHINE
HKLM_Run			Registry	HKEY_LOCAL_MACHINE\SOFTWARE\Micr...
I	903.15	959.87	FileSystem	I:\
J			FileSystem	J:\
K			FileSystem	K:\
L			FileSystem	L:\
LTop	.09	7.60	FileSystem	\\\\$017871\users
M		90.72	FileSystem	M:\
mytemp		74.56	FileSystem	D:\mytemp
N	204.78	90.72	FileSystem	N:\
tmp			Variable	D:\MyTemp
Variable			WSMan	
WSMan				

The Remove-PSDrive removes any of the drives we created for this session.

```
PS C:\Users>
PS C:\Users> Remove-PSDrive LTop
PS C:\Users>
```

Any drives created during the session will disappear when the current session ends. To persist a drive over multiple sessions, use the -persist parameter when creating the drive.

Working with PSDrives

Monday, December 30, 2013
10:18 AM

To list all cmdlets related to working with PSDrives, we use help with a wildcards as shown below. We pipe the output of help to select only the cmdlets of interest.

```
help *item* | select-string 'cmdlet'
```

```
PS C:\Users> PS C:\Users> help *item* | select-string 'cmdlet'

Clear-Item                               Cmdlet    Microsoft.PowerShell.M... Deletes the cont...
Clear-ItemProperty                        Cmdlet    Microsoft.PowerShell.M... Deletes the value...
Copy-Item                                Cmdlet    Microsoft.PowerShell.M... Copies an item f...
Copy-ItemProperty                         Cmdlet    Microsoft.PowerShell.M... Copies a propert...
Get-ChildItem                            Cmdlet    Microsoft.PowerShell.M... Gets the items a...
Get-ControlPanelItem                     Cmdlet    Microsoft.PowerShell.M... Gets control pan...
Get-Item                                  Cmdlet    Microsoft.PowerShell.M... Gets the item at ...
Get-ItemProperty                          Cmdlet    Microsoft.PowerShell.M... Gets the propert...
Invoke-Item                              Cmdlet    Microsoft.PowerShell.M... Performs the def...
Move-Item                                 Cmdlet    Microsoft.PowerShell.M... Moves an item fr...
Move-ItemProperty                         Cmdlet    Microsoft.PowerShell.M... Moves a property...
New-Item                                  Cmdlet    Microsoft.PowerShell.M... Creates a new it...
New-ItemProperty                          Cmdlet    Microsoft.PowerShell.M... Creates a new pr...
Remove-Item                             Cmdlet    Microsoft.PowerShell.M... Deletes the spec...
Remove-ItemProperty                      Cmdlet    Microsoft.PowerShell.M... Deletes the prop...
Rename-Item                             Cmdlet    Microsoft.PowerShell.M... Renames an item...
Rename-ItemProperty                      Cmdlet    Microsoft.PowerShell.M... Renames a proper...
Set-Item                                  Cmdlet    Microsoft.PowerShell.M... Changes the value...
Set-ItemProperty                          Cmdlet    Microsoft.PowerShell.M... Creates or change...
Show-ControlPanelItem                    Cmdlet    Microsoft.PowerShell.M... Opens control pa...
```

Get-Childitem:

The gci cmdlet gets items or child items in one or more specified locations. Items may be files, aliases, functions, registry keys, etc. This cmdlet creates a collection of different types.

This first example list the contents of the windows directory.

```
Get-Childitem c:\windows
```

```
PS C:\Users>
PS C:\Users> Get-ChildItem c:\windows

    Directory: C:\windows

Mode                LastWriteTime     Length Name
----                -----        ---- 
d----   8/22/2013 11:36 AM          addins
d----   8/22/2013 11:36 AM          ADFS
d----   4/17/2015 2:58 AM          AppCompat
d----   5/10/2015 5:40 PM          apppatch
d----   5/17/2015 10:44 AM         AppReadiness
d-r-s   5/13/2015 6:20 AM         assembly
d----   8/22/2013 11:36 AM         Boot
d----   8/22/2013 11:36 AM         Branding
d----   2/11/2015 7:47 AM          Camera
d----   5/25/2015 6:46 PM          CbsTemp
d----   2/11/2015 1:07 AM          CSC
d----   8/22/2013 11:36 AM         Cursors
d----   1/4/2015 5:35 PM          debug
d-r--  8/22/2013 11:36 AM        DesktopTileResources
d----   8/22/2013 11:36 AM        diagnostics
d----   8/22/2013 11:43 AM        DigitalLocker
d---s  8/22/2013 11:36 AM        Downloaded Program Files
d----   1/4/2015 6:57 PM          en-US
d----   2/11/2015 7:47 AM          FileManager
d-r-s   4/29/2015 11:07 PM        Fonts
```

To retrieve all items in the current directory and sub-directories add the -recurse switch.

```
Get-ChildItem c:\windows -recurse
```

-a---	8/22/2013	10:46 AM	0	setuperr.log
-a---	11/4/2014	1:27 AM	128512	sp1wow64.exe
-a---	8/22/2013	2:51 AM	35891	Starter.xml
-a---	8/22/2013	9:25 AM	219	system.ini
-a---	10/28/2014	9:34 PM	54272	twain_32.dll
-a---	8/22/2013	10:47 AM	2723	vmgcoinstall.log
-a---	5/13/2015	5:40 AM	167	win.ini
-a---	6/1/2015	2:35 PM	1981871	WindowsUpdate.log
-a---	10/28/2014	9:53 PM	9728	winhlp32.exe
-a---	6/18/2013	10:54 AM	316640	WMSysPr9.prx
-a---	10/28/2014	10:34 PM	11264	write.exe

Directory: C:\windows\addins

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	6/18/2013	8:21 AM	802 FXSEXT.ecf

Directory: C:\windows\ADFS

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	1/4/2015	6:57 PM	ar
d----	1/4/2015	6:57 PM	bg
d----	1/4/2015	6:57 PM	cs
d----	1/4/2015	6:57 PM	da
d----	1/4/2015	6:57 PM	de
d----	1/4/2015	6:57 PM	el
d----	1/4/2015	6:57 PM	en
l	1/4/2015	6:57 PM	

The above provides some of the same information as the dir command in the legacy cmd shell. However, what makes PowerShell different in more powerful is that the dir command returns text but the get-childitem cmdlet returns file info or directory info objects which may be saved in a variable.

```
$fsobjs = Get-ChildItem c:\windows -Recurse -ErrorAction
SilentlyContinue

$fsobjs | where-object { $_.BaseName -eq 'hosts' }

$fsobjs | ? { $_.BaseName -like 'powershell.exe' }
```

```

PS C:\user>
PS C:\user> $fsobjs = Get-ChildItem c:\windows -Recurse -ErrorAction SilentlyContinue
PS C:\user> $fsobjs | where-object { $_.BaseName -eq 'hosts' }

    Directory: C:\windows\System32\drivers\etc

Mode                LastWriteTime      Length Name
----              -----          ----  --
-a---       6/3/2014 2:48 PM        1182 hosts

    Directory: C:\windows\winsxs\amd64_microsoft-windows-w..nfracture-other_31bf3856ad364

Mode                LastWriteTime      Length Name
----              -----          ----  --
-a---       6/10/2009 5:00 PM         824 hosts

PS C:\user> $fsobjs | ? { $_.BaseName -like 'powershell.exe' }

    Directory: C:\windows\System32\WindowsPowerShell\v1.0\en-US

```

The advantage of saving the objects in a variable is evident in the above examples. In the first pipeline, we search for the "hosts" file by piping the collection of objects stored in \$fsobj into where-object and select only the object whose "BaseName" is equal to "hosts". In the second pipeline, we again search \$fsobj using the where-object alias (?) for the object whose "BaseName" contains "powershell.exe". Since "-like" is a wildcard comparison operation, we select multiple objects. Just like the first search, this search was again done in RAM providing a much faster response. In both cases, the location of the file(s) are displayed on the console since this is the default display for fileinfo objects.

```

PS C:\user> $fsobjs = Get-ChildItem c:\windows -Recurse -ErrorAction SilentlyContinue
PS C:\user> $fsobjs | where-object { $_.BaseName -eq 'hosts' }

    Directory: C:\windows\System32\drivers\etc

Mode                LastWriteTime      Length Name
----              -----        ----
-a---       6/3/2014   2:48 PM        1182 hosts

    Directory: C:\windows\winsxs\amd64_microsoft-windows-w..nfracture-other_31bf3856ad364e35_6.1

Mode                LastWriteTime      Length Name
----              -----        ----
-a---       6/10/2009   5:00 PM         824 hosts

PS C:\user> $fsobjs | ? { $_.BaseName -like 'powershell.exe' }

    Directory: C:\windows\System32\WindowsPowerShell\v1.0\en-US

Mode                LastWriteTime      Length Name
----              -----        ----
-a---       9/27/2013  12:06 AM       11264 powershell.exe.mui

    Directory: C:\windows\SysWOW64\WindowsPowerShell\v1.0\en-US

Mode                LastWriteTime      Length Name
----              -----        ----
-a---       9/26/2013  10:28 PM       11264 powershell.exe.mui

    Directory: C:\windows\winsxs\amd64_microsoft-windows-p..shell-mui.resources_31bf3856ad364e35_6.1

Mode                LastWriteTime      Length Name
----              -----        ----
-a---       7/13/2009  10:26 PM       10240 powershell.exe.mui

```

Below is the equivalent of the above commands using the legacy cmd shell. We observe two fundamental differences, each search required accessing the file system which is significantly slower than the search done in RAM. The more important difference is that we found the file(s) but have no idea as to their location on the file system.

```
C:\user>
C:\user> dir c:\windows /s | find "hosts"
06/03/2014  02:48 PM           1,182 hosts
06/10/2009  05:00 PM           3,683 lmhosts.sam
06/10/2009  05:00 PM           3,683 lmhosts.sam
06/10/2009  05:00 PM           3,683 lmhosts.sam
07/13/2009  09:40 PM          65,536 apphostsvc.dll
11/20/2010  09:25 AM          65,536 apphostsvc.dll
06/10/2009  05:00 PM           824 hosts
07/13/2009  10:57 PM          824 amd64_microsoft-windows-w...nfrastructure-other_31b
ne_6079f415110c0210_hosts_d78df635
07/13/2009  09:14 PM          61,440 apphostsvc.dll
11/20/2010  08:18 AM          61,440 apphostsvc.dll
C:\user> dir c:\windows /s | find "powershell.exe"
09/26/2013  10:13 PM          471,040 powershell.exe
09/27/2013  12:06 AM          11,264 powershell.exe.mui
09/26/2013  08:44 PM          456,704 powershell.exe
09/26/2013  10:28 PM          11,264 powershell.exe.mui
07/13/2009  10:26 PM          10,240 powershell.exe.mui
08/21/2012  11:10 AM          11,264 powershell.exe.mui
09/27/2013  12:06 AM          11,264 powershell.exe.mui
07/13/2009  09:39 PM          473,600 powershell.exe
08/21/2012  08:17 AM          474,624 powershell.exe
09/26/2013  10:13 PM          471,040 powershell.exe
^C
```

Get-Item:

The get-item cmdlet gets, i.e., *instantiates* an object, for a specific item in a specific location. The type of object returned by get-item depends on the item *gotten*. The one important difference between get-childitem and get-item is that get-item is intended from one specific object; where get-childitem is intended to retrieve multiple objects.

In this example we use get-item to create a FileInfo object referencing the budget.xlsx file. We store in the variable \$h for sake of convenience.

```
$h = get-item d:\MyTemp\budget.xlsx
$h | gm
```

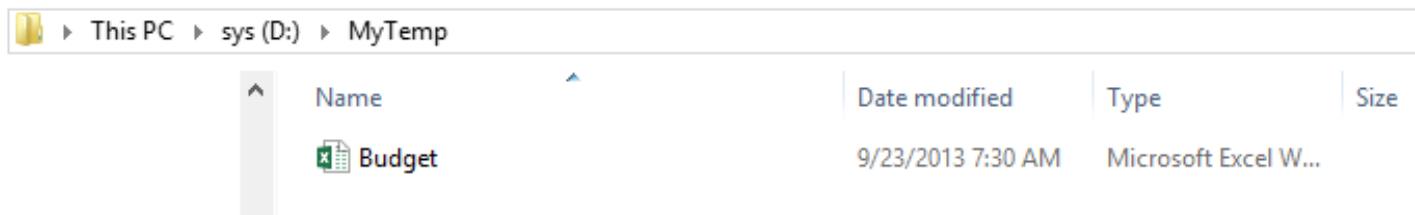
```

PS C:\Users>
PS C:\Users> $h = get-item d:\MyTemp\budget.xlsx
PS C:\Users> $h | gm

```

Type Name: System.IO.FileInfo

Name	MemberType	Definition
---	-----	-----
Mode	CodeProperty	System.String Mode{get=Mode;}
AppendText	Method	System.IO.StreamWriter AppendText()
CopyTo	Method	System.IO.FileInfo CopyTo(string destFileName)
Create	Method	System.IO FileStream Create()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef CreateObjRef(string)
CreateText	Method	System.IO.StreamWriter CreateText()
Decrypt	Method	void Decrypt()
Delete	Method	void Delete()
Encrypt	Method	void Encrypt()
Equals	Method	bool Equals(System.Object obj)
GetAccessControl	Method	System.Security.AccessControl.FileSecurity GetAccessControl()
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeService()
GetObjectData	Method	void GetObjectData(System.Runtime.Serialization.SerializationInfo info, System.Runtime.Serialization.StreamingContext context)
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object InitializeLifetimeService()
MoveTo	Method	void MoveTo(string destFileName)
Open	Method	System.IO FileStream Open(System.String path, FileMode mode)
OpenRead	Method	System.IO FileStream OpenRead()
OpenText	Method	System.IO.StreamReader OpenText()
OpenWrite	Method	System.IO FileStream OpenWrite()



With \$h holding the object referencing the hosts file, we may now use the properties and methods of the object to act on the file. We may display the associated properties of the object as shown below. We use the Write-Host cmdlet to display multiple properties to the console. One of the advantages of using write-host is that we may specify a color for the items displayed. Finally, we may copy of the file using the method CopyTo.

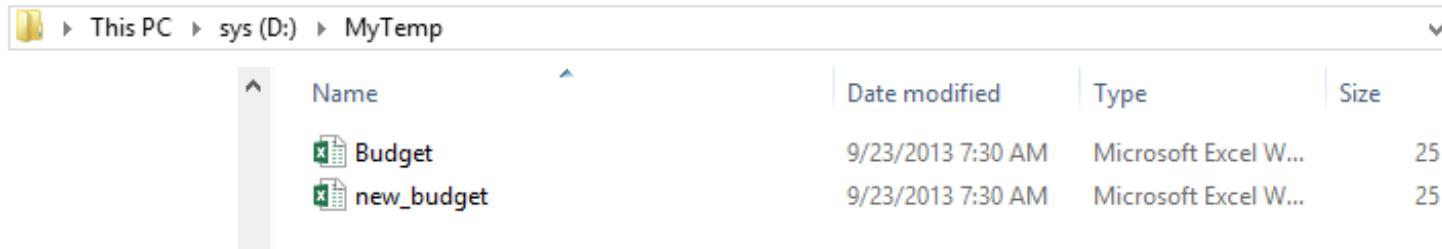
```
$h.Name
```

```
write-host $h.BaseName $h.length $h.LastAccessTime -ForegroundColor yellow  
  
$h.CopyTo('d:\mytemp\new_budget.xlsx')
```

```
PS C:\Users>  
PS C:\Users> $h.Name  
budget.xlsx  
PS C:\Users>  
PS C:\Users> write-host $h.BaseName $h.Length $h.LastAccessTime -ForegroundColor yellow  
budget 24687 6/4/2015 8:49:52 AM  
PS C:\Users>  
PS C:\Users> $h.CopyTo('d:\mytemp\new_budget.xlsx')  


| Mode  | LastWriteTime     | Length | Name            |
|-------|-------------------|--------|-----------------|
| -a--- | 9/23/2013 7:30 AM | 24687  | new_budget.xlsx |

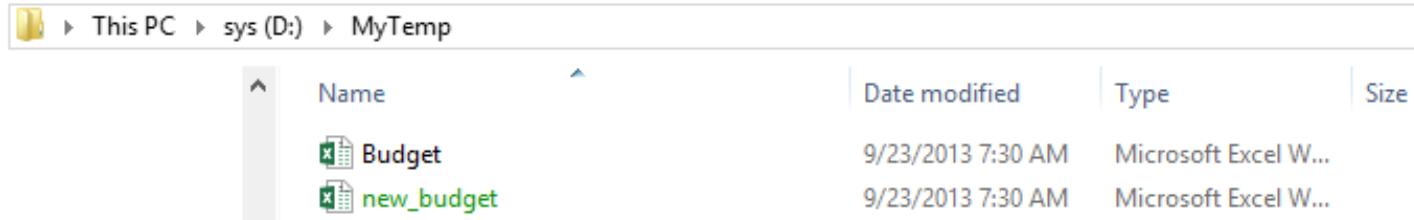

```



We may also encrypt the file using the Encrypt method.

```
$new_budget = get-item d:\MyTemp\new_budget.xlsx  
  
$new_budget.Encrypt()
```

```
PS C:\Users>  
PS C:\Users> $new_budget = get-item d:\MyTemp\new_budget.xlsx  
PS C:\Users>  
PS C:\Users> $new_budget.Encrypt()  
PS C:\Users>
```



Executing get-item against a directory results in the instantiation of a DirectoryInfo object. Notice how PowerShell *recasts* (*changed the object type*) the variable \$h from a FileInfo to a DirectoryInfo object. Dynamic *typing* of variables is one of the more useful features of PowerShell.

```
$h = get-item C:\Windows\System32\drivers\etc  
$h | gm
```

```
PS C:\user> $h = get-item C:\Windows\System32\drivers\etc  
PS C:\user> $h | gm  
  
TypeName: System.IO.DirectoryInfo  


| Name                     | MemberType   | Definition                                 |
|--------------------------|--------------|--------------------------------------------|
| ---                      | -----        | -----                                      |
| Mode                     | CodeProperty | System.String Mode{get=Mode; }             |
| Create                   | Method       | void Create(), void Create(System.Security |
| CreateObjRef             | Method       | System.Runtime.Remoting.ObjRef CreateOb    |
| CreateSubdirectory       | Method       | System.IO.DirectoryInfo CreateSubdirect    |
| Delete                   | Method       | void Delete(), void Delete(bool recursi    |
| EnumerateDirectories     | Method       | System.Collections.Generic.IEnumerable[    |
| EnumerateFiles           | Method       | System.Collections.Generic.IEnumerable[    |
| EnumerateFileSystemInfos | Method       | System.Collections.Generic.IEnumerable[    |
| Equals                   | Method       | bool Equals(System.Object obj) .           |


```

Running the cmdlet against a registry key results in a RegistryKey object.

```
$h = get-item HKLM:\software\microsoft\Windows\CurrentVersion\run  
$h | gm
```

```

PS C:\user>
PS C:\user> $h = get-item HKLM:\software\microsoft\windows\CurrentVersion\run
PS C:\user> $h | get-member

TypeName: Microsoft.Win32.RegistryKey

Name          MemberType      Definition
----          -----
Close         Method         void Close()
CreateObjRef  Method         System.Runtime.Remoting.ObjRef CreateObjR...
CreateSubKey   Method         Microsoft.Win32.RegistryKey CreateSubKey(...
DeleteSubKey  Method         void DeleteSubKey(string subkey), void De...
DeleteSubKeyTree Method        void DeleteSubKeyTree(string subkey), voi...
DeleteValue   Method         void DeleteValue(string name), void Delete...
Dispose       Method         void Dispose(), void IDisposable.Dispose()
Equals        Method         bool Equals(System.Object obj)
Flush         Method         void Flush()

```

Once we have the object stored in \$h, we may inspect some of the properties as shown below. We inspect the default display of the properties by just entering the variable name at the command line. This results in the values stored in the registry key "run". Note, that this is a multi-valued property, i.e., it contains values that are properties. We may also inspected other properties such as "Name" or "PSProvider". Finally, we execute the "GetValue" method. This method allows us to retrieve the value of the property "MSC" which is one of the properties stored in Run.

```

PS C:\Users>
PS C:\Users> $h

    Hive: HKEY_LOCAL_MACHINE\software\microsoft\Windows\CurrentVersion

Name          Property
----          -----
run
ATIModeChange : Ati2mdxx.exe
IAAnotif      : C:\Program Files (x86)\Intel\Intel Matrix
DellControlPoint : "c:\Program Files\Dell\Dell ControlPoint\Se
USCService    : C:\Program Files\Microsoft Security Client\Se
MSC           : "c:\Program Files\Broadcom\BACS\BacsTray.exe"
bacstray       : C:\Program Files\Broadcom\BACS\BacsTray.exe
SonicWALLNetExtender : C:\Program Files (x86)\SonicWALL\SSL-VPN\N

PS C:\Users> $h.Name
HKEY_LOCAL_MACHINE\software\microsoft\Windows\CurrentVersion\run
PS C:\Users>
PS C:\Users> $h.PSPProvider

Name          Capabilities
----          -----
Registry      ShouldProcess, Transactions

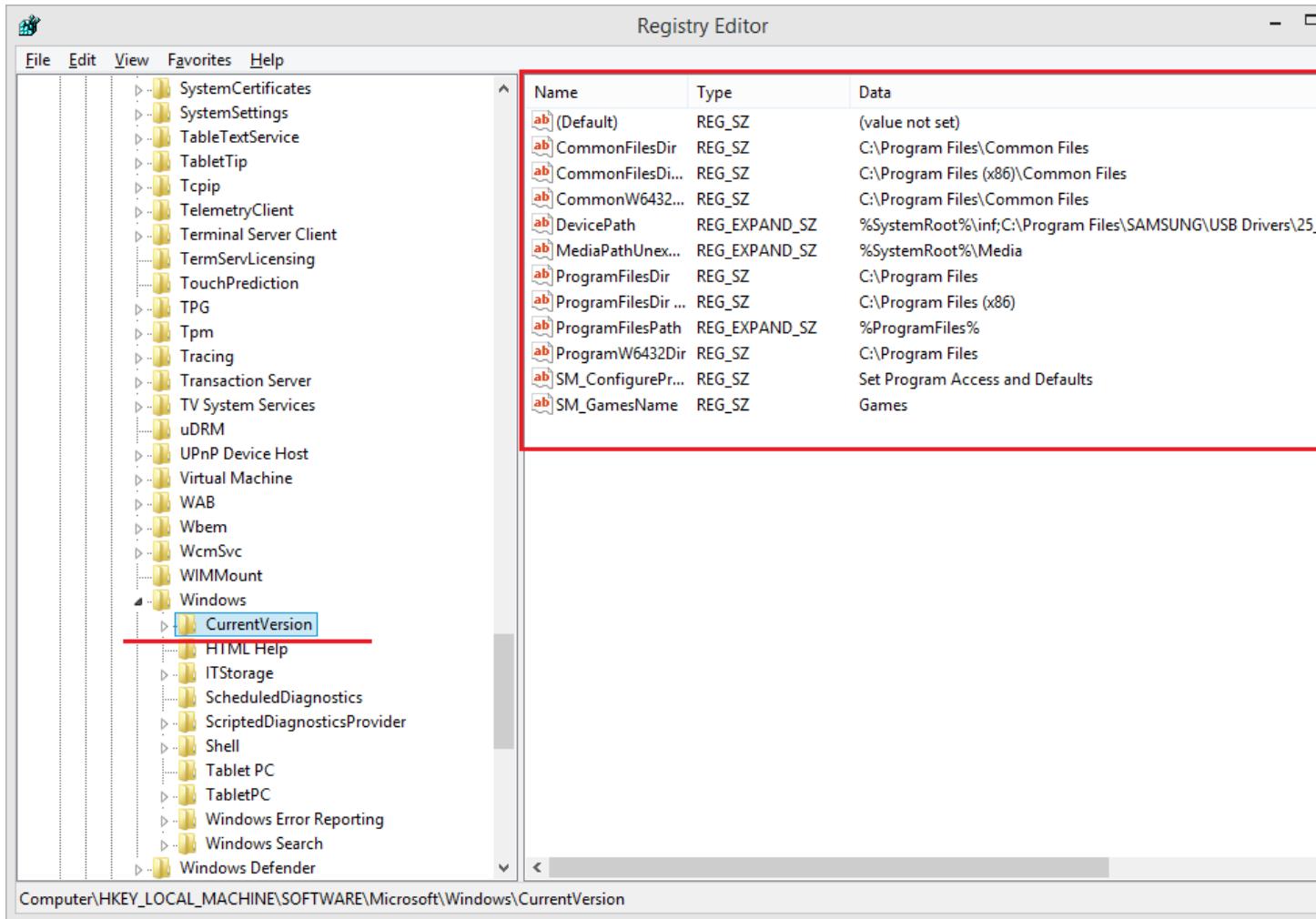
PS C:\Users> $h.GetValue('MSC')
"c:\Program Files\Microsoft Security Client\msseces.exe" -hide -runkey
PS C:\Users>

```

Get-ItemProperty:

The get-itemproperty cmdlet retrieves the properties of an item. For the file system, get-itemproperty and get-item return identical objects. For the Registry, get-itemproperty is the better cmdlet since it returns a PowerShell custom object with properties derived from the registry key or value.

The illustration below shows the Values of the CurrentVersion key.



We illustrate the difference between these two cmdlets below. Notice that the variable \$gp has properties, i.e., NoteProperties, that are derived directly from the registry key values. The \$gi variable, which is a RegistryKey object, does not have those properties.

```
$gi = Get-Item -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion

$gp = Get-ItemProperty -Path
HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion
```

```

PS C:\Users>
PS C:\Users> $gi = Get-Item -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion
PS C:\Users> $gp = Get-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion
PS C:\Users>
PS C:\Users> $gi | gm | select -first 10

    TypeName: Microsoft.Win32.RegistryKey

Name          MemberType  Definition
----          --          --
Close         Method     void Close()
CreateObjRef  Method     System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
CreateSubKey   Method     Microsoft.Win32.RegistryKey CreateSubKey(string subkey), Microsoft.Win32.RegistryKey CreateSubKey(string subkey, Microsoft.Win32.RegistryKeyPermissionCheck permission)
DeleteSubKey   Method     void DeleteSubKey(string subkey), void DeleteSubKey(string subkey, Microsoft.Win32.RegistryKeyPermissionCheck permission)
DeleteSubKeyTree Method     void DeleteSubKeyTree(string subkey), void DeleteSubKeyTree(string subkey, Microsoft.Win32.RegistryKeyPermissionCheck permission)
DeleteValue    Method     void DeleteValue(string name), void DeleteValue(string name, Microsoft.Win32.RegistryValueKind kind)
Dispose       Method     void Dispose(), void IDisposable.Dispose()
Equals        Method     bool Equals(System.Object obj)
Flush         Method     void Flush()
GetAccessControl Method     System.Security.AccessControl.RegistrySecurity GetAccessControl()

```



```

PS C:\Users> $gp | gm | select -first 10

    TypeName: System.Management.Automation.PSCustomObject

Name          MemberType  Definition
----          --          --
Equals        Method     bool Equals(System.Object obj)
GetHashCode   Method     int GetHashCode()
GetType       Method     type GetType()
ToString      Method     string ToString()
CommonFilesDir NoteProperty System.String CommonFilesDir=C:\Program Files\Common Files
CommonFilesDir (x86) NoteProperty System.String CommonFilesDir (x86)=C:\Program Files (x86)\Common Files
CommonW6432Dir NoteProperty System.String CommonW6432Dir=C:\Program Files\Common Files
DevicePath    NoteProperty System.String DevicePath=C:\Windows\inf;C:\Program Files\Device Path
MediaPathUnexpanded NoteProperty System.String MediaPathUnexpanded=C:\Windows\Media
ProgramFilesDir NoteProperty System.String ProgramFilesDir=C:\Program Files

```

Displaying the contents of the variables clearly shows the differences. The variable \$gi has one multi-valued property where \$gp has properties for every value of the key CurrentVersion.

```

PS C:\Users>
PS C:\Users> $gp

ProgramFilesDir          : C:\Program Files
CommonFilesDir           : C:\Program Files\Common Files
ProgramFilesDir (x86)     : C:\Program Files (x86)
CommonFilesDir (x86)      : C:\Program Files (x86)\Common Files
CommonW6432Dir            : C:\Program Files\Common Files
ProgramW6432Dir           : C:\Program Files
MediaPathUnexpanded       : C:\Windows\Media
DevicePath                : C:\Windows\inf;C:\Program Files\SAMSUNG\USB Drivers\25_escape
ProgramFilePath           : C:\Program Files
SM_GamesName              : Games
SM_ConfigureProgramsName  : Set Program Access and Defaults
PSPath                     : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWA
                           sion
PSParentPath               : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWA
                           sion
PSChildName                : CurrentVersion
PSDrive                      : HKLM
PSPProvider                  : Microsoft.PowerShell.Core\Registry

PS C:\Users> $gi

Hive: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows

Name          Property
----          -----
CurrentVersion
ProgramFilesDir          : C:\Program Files
CommonFilesDir           : C:\Program Files\Common Files
ProgramFilesDir (x86)     : C:\Program Files (x86)
CommonFilesDir (x86)      : C:\Program Files (x86)\Common Files
CommonW6432Dir            : C:\Program Files\Common Files
ProgramW6432Dir           : C:\Program Files
MediaPathUnexpanded       : C:\Windows\Media
DevicePath                : C:\Windows\inf;C:\Program File
ProgramFilePath           : C:\Program Files
SM_GamesName              : Games
SM_ConfigureProgramsName  : Set Program Access and Defaults

```

Retrieving a specific value requires the method `GetValue` of the variable `$gi` created by `get-item` where we only need to reference the property name using the variable `$gp`.

```

$gi.GetValue('ProgramFilesDir')

$gp.ProgramFilesDir

```

```
PS C:\Users>
PS C:\Users> $gi.GetValue('ProgramFilesDir')
C:\Program Files
PS C:\Users>
PS C:\Users> $gp.ProgramFilesDir
C:\Program Files
PS C:\Users>
```

New-Item

As the name implies, the new-item cmdlet creates a new item in the corresponding PSDrive. The type of item created depends on what PSDrive on which the item is created. This cmdlet may be used to create files, directory, registry keys and values, and other items.

Below we create a new text file called new_file.txt and add some text to the file. In the cmdlet below the metacharacter `n indicates a new-line character. This will create the file with four lines of text. Using double quotes to delimit the value string is required. Double quotes cause PowerShell to inspect the contents of the string and expand variables, sub-expressions, and metacharacters. If you were to use single quotes, the file would contain one line with the text shown.

```
new-item new_file.txt -itemtype file -value "line 1`nLine 2`nLine 3`nLine 4`n"
get-content new_file.txt
```

```
PS C:\Users>
PS C:\Users> new-item new_file.txt -itemtype file -value "line 1`nLine 2`nLine 3`nLine 4`n"
Directory: C:\Users

Mode                LastWriteTime     Length Name
----                -----          ---- 
-a---       6/9/2015   3:55 PM        28 new_file.txt

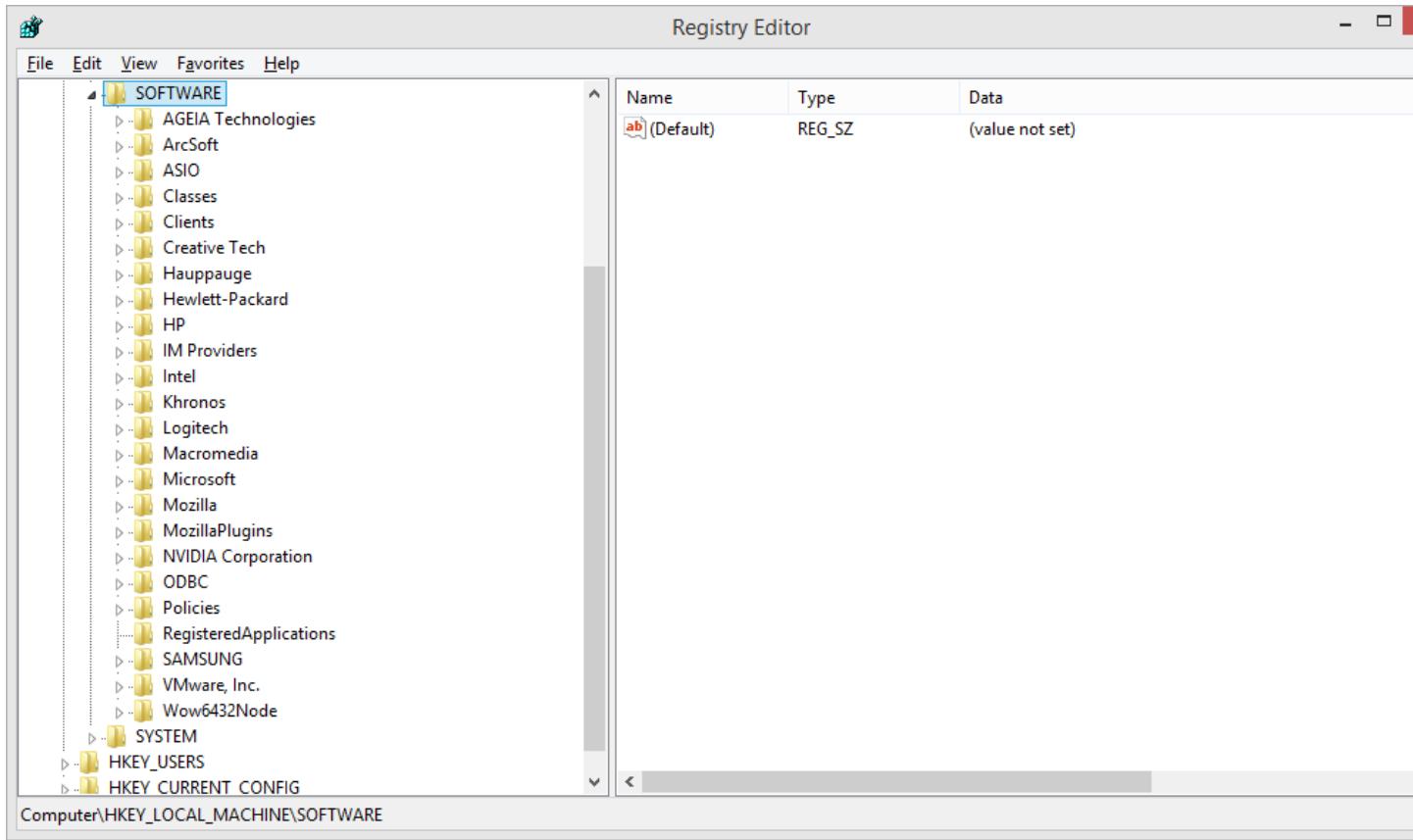
PS C:\Users> Get-Content .\new_file.txt
Line 1
Line 2
Line 3
Line 4
PS C:\Users>
```

Creating a directory simply requires change the -ItemType parameter.

```
new-item mydir -itemtype directory  
get-childitem
```

```
PS C:\Users>  
PS C:\Users> Get-ChildItem  
  
Directory: C:\Users  
  
Mode LastWriteTime Length Name  
---- -- -- -- --  
d--- 4/16/2015 11:09 PM mike  
d-r-- 2/12/2015 9:09 AM Public  
-a--- 2/24/2015 7:19 PM file.txt  
-a--- 3/16/2015 2:31 PM hello.ps1  
-a--- 6/9/2015 3:55 PM new_file.txt  
-a--- 3/16/2015 3:36 PM script.ps1  
  
PS C:\Users> new-item mydir -itemtype directory  
  
Directory: C:\Users  
  
Mode LastWriteTime Length Name  
---- -- -- -- --  
d--- 6/10/2015 3:31 PM mydir  
  
PS C:\Users> get-childitem  
  
Directory: C:\Users  
  
Mode LastWriteTime Length Name  
---- -- -- -- --  
d--- 4/16/2015 11:09 PM mike  
d--- 6/10/2015 3:31 PM mydir  
d-r-- 2/12/2015 9:09 AM Public  
-a--- 2/24/2015 7:19 PM file.txt  
-a--- 3/16/2015 2:31 PM hello.ps1  
-a--- 6/9/2015 3:55 PM new_file.txt  
-a--- 3/16/2015 3:36 PM script.ps1
```

The current state of the registry is shown below. The example below creates a new registry key called ps_made_key in the path HKLM:\Software.



Note: when modifying the registry, the PowerShell console must run with elevated privileges., i.e., "Run as Administrator".

```
new-item hklm:\software\ps_made_key
```

```
get-childitem hklm:\software
```

```
PS C:\Windows\system32> PS C:\Windows\system32> new-item hklm:\software\ps_made_key

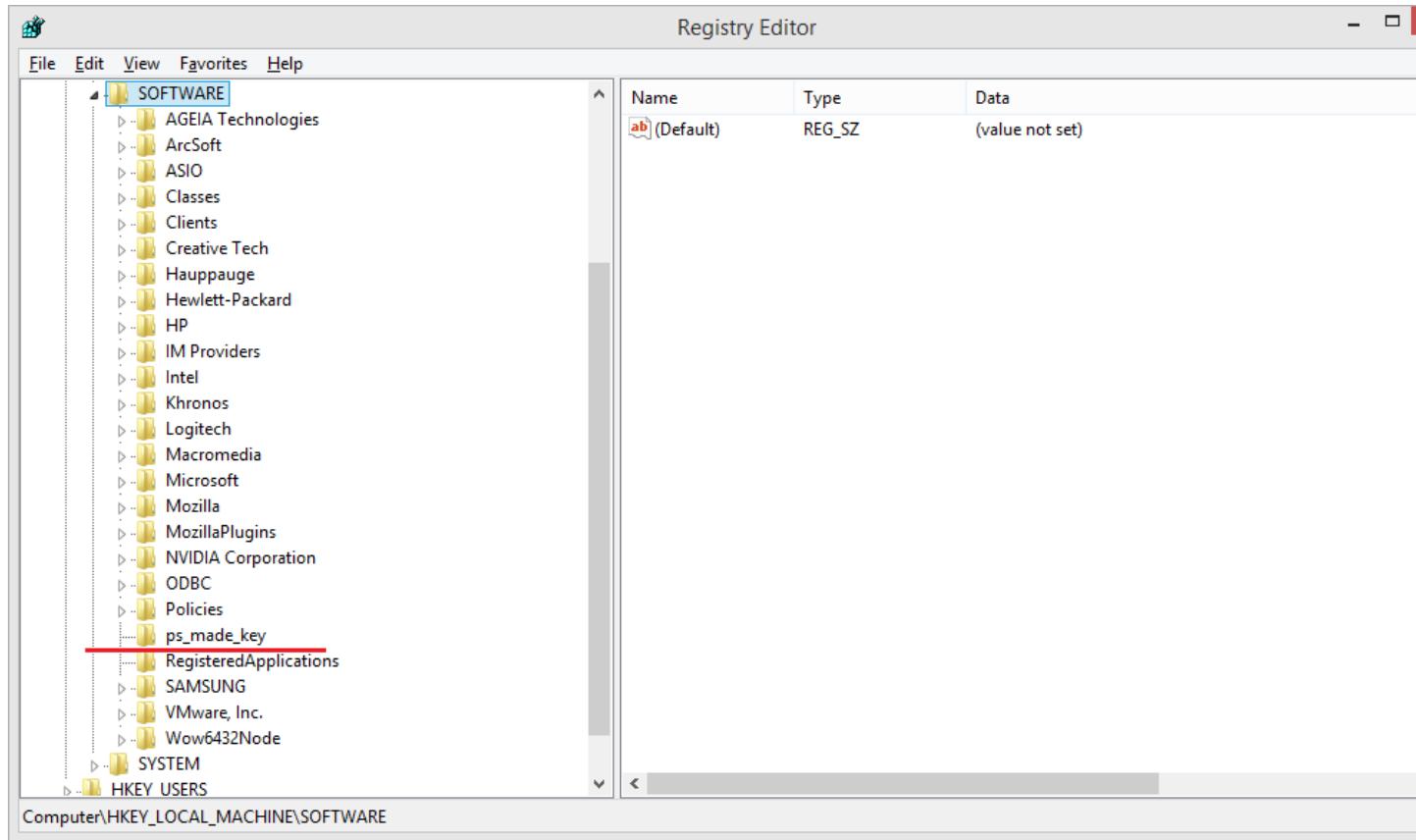
    Hive: HKEY_LOCAL_MACHINE\software

Name          Property
----          -----
ps_made_key

PS C:\Windows\system32> Get-ChildItem hklm:\software

    Hive: HKEY_LOCAL_MACHINE\software

Name          Property
----          -----
AGEIA Technologies
ArcSoft
ASTO
Classes
Clients
Creative Tech
Hauppauge
Hewlett-Packard
HP
IM Providers
Intel
Khrongos
Logitech
Macromedia
Microsoft
Mozilla
MozillaPlugins
NVIDIA Corporation
ODBC
Policies
ps_made_key
RegisteredApplications          Paint : SOFTWARE\Microsoft\Windows\CurrentVersion\Applets\
```



New-ItemProperty

The `new-itemproperty` cmdlet creates a new property for an item. This cmdlet is useful when creating Registry values. The interface for this cmdlet is not implemented by the File System Provider.

```
new-itemproperty -path hklm:\software\ps_made_key -name PSKey1 `  
    -propertytype 'String' -value 'A PS created value'  
  
get-childitem hklm:\software
```

```

PS C:\Windows\system32>
PS C:\Windows\system32> new-itemproperty -path hklm:\software\ps_made_key -name PSKey1
>> -propertytype 'String' -value 'A PS created value'
>>

PSKey1      : A PS created value
PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\software\ps_made_
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\software
PSChildName : ps_made_key
PSDrive     : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry

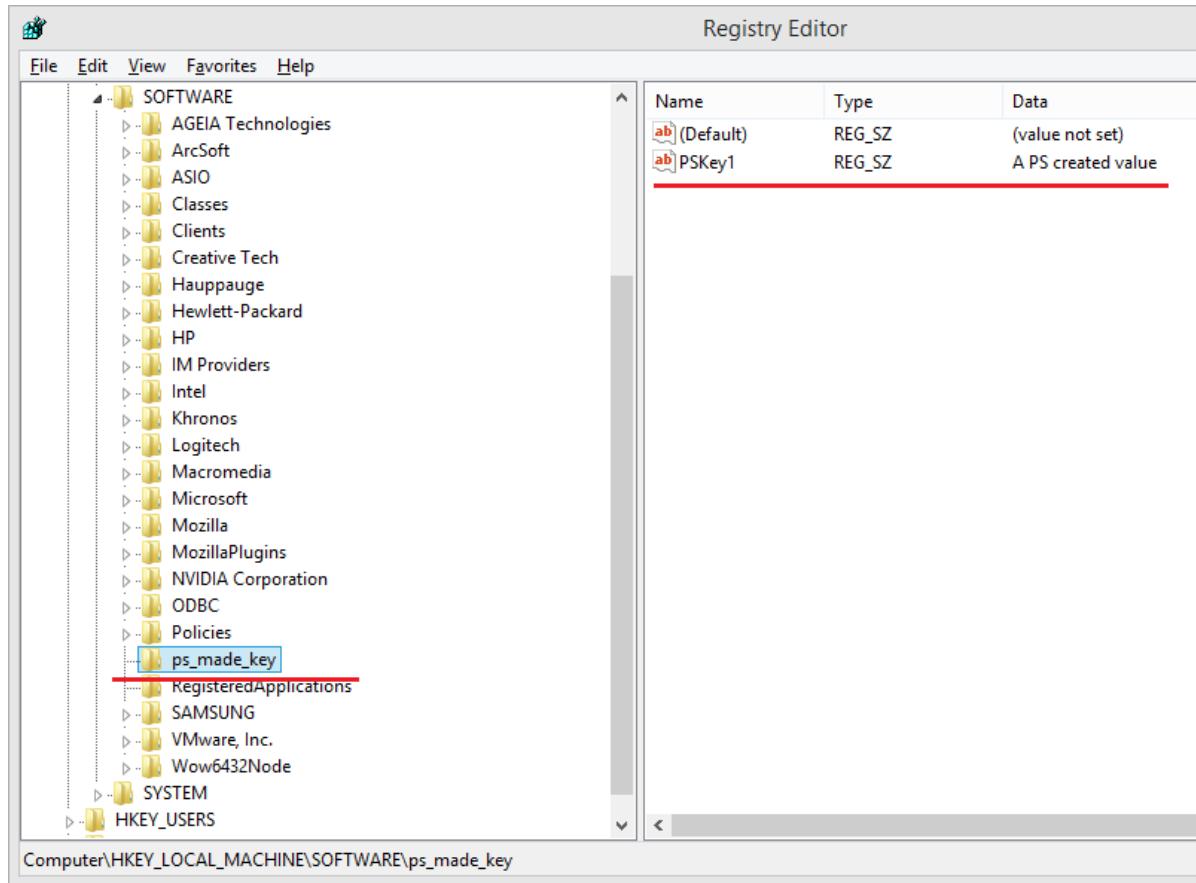
PS C:\Windows\system32> get-childitem hklm:\software

    Hive: HKEY_LOCAL_MACHINE\software



| Name                   | Property                                                                                                                             |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| AGEIA Technologies     | HwSelection : GPU<br>PhysX Version : 9150428<br>PhysXCore Path : C:\Program Files (x86)\NVIDIA Corporation<br>PhysX BuildCL : 1      |
| ArcSoft                |                                                                                                                                      |
| ASIO                   |                                                                                                                                      |
| Classes                | (default) :                                                                                                                          |
| Clients                |                                                                                                                                      |
| Creative Tech          |                                                                                                                                      |
| Hauppauge              |                                                                                                                                      |
| Hewlett-Packard        |                                                                                                                                      |
| HP                     |                                                                                                                                      |
| IM Providers           |                                                                                                                                      |
| Intel                  |                                                                                                                                      |
| Khronos                |                                                                                                                                      |
| Logitech               |                                                                                                                                      |
| Macromedia             |                                                                                                                                      |
| Microsoft              |                                                                                                                                      |
| Mozilla                |                                                                                                                                      |
| MozillaPlugins         |                                                                                                                                      |
| NVIDIA Corporation     |                                                                                                                                      |
| ODBC                   |                                                                                                                                      |
| Policies               |                                                                                                                                      |
| ps_made_key            | PSKey1 : A PS created value                                                                                                          |
| RegisteredApplications | Paint : SOFTWARE\Microsoft\Windows\CurrentVersion\Applets\Paint<br>Windows Search : Software\Microsoft\Windows\CurrentVersion\Search |


```



An existing registry key may also be updated as shown in the following example. In this case, we want to add a property to the HKLM:\Software\Microsoft\Windows\Currentversion\Run key.

The sequences of commands do the following: 1) navigate to the key HKLM:\Software\Microsoft\Windows\Currentversion; 2) use get-item to store the Run key into \$run; 3) Display the properties in the key; 4) Pipe \$run into new-itemproperty to add the property hello with a value of 'c:\hello.bat'; 5) Display the properties again which now shows the new property; 6) Use remove-itemproperty to remove the hello property just added.

```

PS C:\Windows\system32>
PS C:\Windows\system32> set-location HKLM:\software\Microsoft\Windows\CurrentVersion\
PS HKLM:\software\Microsoft\Windows\CurrentVersion\>
PS HKLM:\software\Microsoft\Windows\CurrentVersion\> $run = get-item run
PS HKLM:\software\Microsoft\Windows\CurrentVersion\>
PS HKLM:\software\Microsoft\Windows\CurrentVersion\> $run | ft -autosize

    Hive: HKEY_LOCAL_MACHINE\software\Microsoft\Windows\CurrentVersion

Name Property
---- -----
run   ATIModeChange      : Ati2mdxx.exe
IAAnotif          : C:\Program Files (x86)\Intel\Intel Matrix Storage Manager\jaannotif.exe
DellControlPoint  : "C:\Program Files\Dell\ControlPoint\ControlPoint.exe"
MSC              : "C:\Program Files\Microsoft Security Client\msseces.exe" -hide -runkey
bacstray          : C:\Program Files\Broadcom\BACS\BacsTray.exe
SonicWALLNetExtender : C:\Program Files (x86)\SonicWALL\SSL-VPN\NetExtender\NEGUI.exe -hideGUI -cle

PS HKLM:\software\Microsoft\Windows\CurrentVersion\> $run | New-ItemProperty -name hello -value 'c:\hell
hello      : c:\hello.bat
PSPath     : Microsoft.PowerShell.Core\Registry:::HKEY_LOCAL_MACHINE\software\Microsoft\Windows\Current
PSParentPath: Microsoft.PowerShell.Core\Registry:::HKEY_LOCAL_MACHINE\software\Microsoft\Windows\Current
PSChildName: run
PSProvider  : Microsoft.PowerShell.Core\Registry

PS HKLM:\software\Microsoft\Windows\CurrentVersion\> $run | ft -autosize

    Hive: HKEY_LOCAL_MACHINE\software\Microsoft\Windows\CurrentVersion

Name Property
---- -----
run   ATIModeChange      : Ati2mdxx.exe
IAAnotif          : C:\Program Files (x86)\Intel\Intel Matrix Storage Manager\jaannotif.exe
DellControlPoint  : "C:\Program Files\Dell\ControlPoint\ControlPoint.exe"
MSC              : "C:\Program Files\Microsoft Security Client\msseces.exe" -hide -runkey
bacstray          : C:\Program Files\Broadcom\BACS\BacsTray.exe
SonicWALLNetExtender : C:\Program Files (x86)\SonicWALL\SSL-VPN\NetExtender\NEGUI.exe -hideGUI -cle
hello      : c:\hello.bat

PS HKLM:\software\Microsoft\Windows\CurrentVersion\> $run | Remove-ItemProperty -name hello
PS HKLM:\software\Microsoft\Windows\CurrentVersion\>

```

Set-Item

The Microsoft Technet describes this cmdlet as "changes the value of an item..." This cmdlet is useful when changing items that have values like environmental variables, aliases, and registry values. The set-item cmdlet may not be used with PSDrive objects that don't have an associated values like file or directories.

```

set-item -path alias:ed -value 'C:\Program Files
(x86)\Notepad++\notepad++.exe'

get-alias | where-object { $_.definition -like '*notep*' }

```

```
PS C:\Users> set-item -path alias:ed -value 'C:\Program Files (x86)\Notepad++\notepad++'
PS C:\Users> get-alias | where-object { $_.definition -like '*notep*' }
CommandType      Name          ModuleName
----           ----
Alias           ed -> notepad++.exe
```

The following illustrates using set-item to change a variable value. The set-item syntax below is silly since it is much easier to write `$a = 0`.

```
PS C:\Users>
PS C:\Users> $a = 'abcdef'
PS C:\Users>
PS C:\Users> $a.GetType()
IsPublic IsSerial Name                                     BaseType
-----  ----- 
True      True    String                                 System.Object

PS C:\Users> set-item -path variable:a -value 0
PS C:\Users>
PS C:\Users> $a.GetType()
IsPublic IsSerial Name                                     BaseType
-----  ----- 
True      True    Int32                                System.ValueType
```

We may also change environment variables using `set-item`. In the example below, we append a directory to the `PATH` environment variable. Changes made to environment variables do not persist beyond the session in which the change was made.

```
$new_path = (get-item env:path).Value + ' ;C:\Program Files  
(x86)\Microsoft Office\Office15'  
set-item -path env:path -value $new_path
```

Set-ItemProperty

We use *set-itemproperty* to change a property of a multi-value object like a registry key.

```
set-itemproperty -path hklm:\software\ps_made_key `  
-name PSKey1 -value 'A new PS value'
```

```
get-childitem hklm:\software
```

```
PS C:\Users>
PS C:\Users> set-itemproperty -path hklm:\software\ps_made_key `>>>           -name PSKey1 -value 'A new PS value' ...
PS C:\Users> get-childitem hklm:\software

Hive: HKEY_LOCAL_MACHINE\software

Name          Property
----          -----
Alps
Classes
Clients
Dell
InstalledOptions
Intel          (default) :
Khrongos
Macromedia
Microsoft
Mozilla
ODBC
OEM
Partner
Policies
ps_made_key    PSKey1 : A new PS value
RegisteredApplications   File Explorer      :
                                         : SOFTWARE\Microsoft\Windows\CurrentVersion\Expl
```

The Environment Drive

Friday, June 19, 2015
4:09 PM

PowerShell virtualizes the Windows environment variables in the env: drive. As shown later, environment variables may be modified or created within PowerShell similar to the legacy cmd shell. However any new environment variables created or any variables altered are local to the session. This means they exist only while the session is active. Once the session is terminated, any new environment variables disappear and modified environment variables modified revert back to the value before the session began.

Changing Environment Variables using cmd:

When a command is entered on the command line in the cmd shell, Windows first looks in the current directory for a .exe or .bat file of that name. Not finding either, Windows searches the directories for an executable of that name in the directories listed in the environment variable *Path*. The *Path* value is a list of directories where Windows searches when the executable name is entered in the command line in command (cmd) windows or the PowerShell console.

```
C:\Users>
C:\Users> set
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\mike\AppData\Roaming
CommonProgramFiles=C:\Program Files\Common Files
CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
CommonProgramW6432=C:\Program Files\Common Files
COMPUTERNAME=EL-CID
ComSpec=C:\Windows\system32\cmd.exe
FP_NO_HOST_CHECK=NO
HOMEDRIVE=C:
HOMEPATH=\Users\mike
LOCALAPPDATA=C:\Users\mike\AppData\Local
LOGONSERVER=\\"MicrosoftAccount
NUMBER_OF_PROCESSORS=12
OS=Windows_NT
Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\Skype\Phone\;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.UBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE=AMD64
PROCESSOR_IDENTIFIER=Intel64 Family 6 Model 45 Stepping 7, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=2d07
ProgramData=C:\ProgramData
ProgramFiles=C:\Program Files
ProgramFiles(x86)=C:\Program Files (x86)
ProgramW6432=C:\Program Files
PROMPT=$p$g$s
PSModulePath=C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
PUBLIC=C:\Users\Public
SESSIONNAME=Console
SystemDrive=C:
SystemRoot=C:\Windows
TEMP=d:\usrtemp
TMP=d:\usrtemp
USERDOMAIN=EL-CID
USERDOMAIN_ROAMINGPROFILE=EL-CID
USERNAME=mike
USERPROFILE=C:\Users\mike
windir=C:\Windows
```

Given the above Path, entering *notepad* on the command line would cause Notepad to execute since Windows would find the executable *notepad.exe* in the c:\windows\system32 which is contained in the Path variable. However, entering

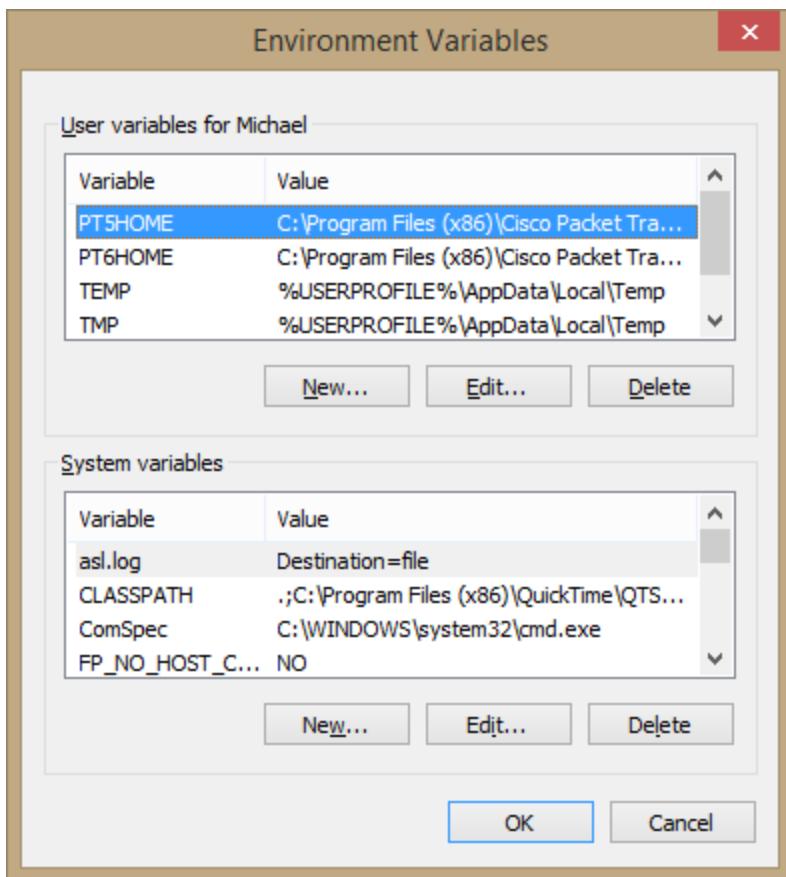
winword (MS Word) on the command line results in an error because *winword.exe* is located in the directory C:\Program Files (x86)\Microsoft Office\Office15 which is not listed in the Path environment variable.

```
C:\Users>
C:\Users> winword
'winword' is not recognized as an internal or external command,
operable program or batch file.
C:\Users>
```

We could execute *winword.exe* by eliminating any ambiguity in the executable name by specifying the full pathname to the executable as below. However this will get rather tedious quickly.

```
C:\Users>
C:\Users> "C:\Program Files (x86)\Microsoft Office\Office15\winword"
```

We may solve this problem by adding the path to *winword.exe* to the Path variable. The traditional way of doing this requires using the Windows GUI to open the Computer properties windows then opening the Advanced System Settings window then opening the Environment Variables window to finally arrive at the window that allows create a new environment variable. However, many times we just want to temporarily alter the Path for the current session and not modify the Path permanently.



The cmd shell provides a simpler method to alter the Path variable dynamically. Using the example above we may temporarily alter the Path for the current session by appending the fully qualified name of the directory that contains the winword executable to the current value of the Path environment variable.

```
C:\Users>
C:\Users> echo %path%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerS...
6>\Skype\Phone\;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common
C:\Users> set Path=%Path%;C:\Program Files (x86)\Microsoft Office\Office15
C:\Users> echo %path%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerS...
6>\Skype\Phone\;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files (...
```

The above set command takes the current value of the path variable %PATH% and appends to it the path to the location where winword.exe exists. The resultant string is stored back in the PATH environment variable. The quirky syntax, when to use % and when not, has historically been problematic in the cmd shell.

Changing Environment Variables using PowerShell:

PowerShell uses the environment variables in a similar manner as the cmd shell. However PowerShell takes a different approach in altering environment variables. PowerShell provides the abstracted drive called `env:` that provides access to the environment variables. We use the cmdlet `Get-ChildItem` to retrieve the variables;

```
Get-ChildItem env:
```

PowerShell also provides a pseudo multi-valued variable \$env that allows direct access to any other environment variables. We may reference an environmental variable by prepending the name with \$env as shown below.

```
PS C:\Users>
PS C:\Users> write-host $env:computername `n
>>> $env:number_of_processors -foregroundcolor red
MINK 4
```

Having this multi-valued variable allows us to conveniently modify an environment variable by using an assignment statement. The example below demonstrates appending an additional pat to the environment variable Path.

```
$env:path = $env:path + ';C:\Program Files (x86)\Microsoft
Office\Office15'
```

The above syntax retrieves the current path variable, \$env:path then concatenates (+) the string '`;C:\Program Files (x86)\Microsoft Office\Office15`' the resultant string is assigned (=) back to \$env:path. After this operation is complete, winword.exe, or any other executable in `C:\Program Files (x86)\Microsoft Office\Office15` may be executed by entering the executable name in the current PowerShell session. The operation result of the above PowerShells statement is shown below. The cmdlet get-childitem retrieves all of the items (environment variables) in the env: drive. Each object is sequentially piped to where-object which selects only the path item which is subsequently piped to format-table for display on the console.

```
PS C:\user> get-childitem env: | where-object { $_.Name -eq 'Path' } | format-table -wrap
Name Value
---- -----
Path %SystemRoot%\system32\WindowsPowerShell\v1.0\;C:\ProgramData\Oracle\Java\javapath;C:\Shared\Windows_Live;C:\oracle\11.2.0\client_1;C:\Windows\system32;C:\Windows;C:\Windows\Cryptosystems\NTRU_TCG_Software_Stack\bin\;C:\Program Files\NTRU_Cryptosystems\NTRU_TCG_Software_Stack\bin\;C:\Program Files\NTRU_TCG_Software_Stack\bin\;C:\Program Files (x86)\Common Files\Roxio Shared\DLLShared\;C:\Program Files (x86)\Common Files\Roxio Live\Shared;C:\Users\mgalea\NDI\ps_scripts;C:\Program Files (x86)\isc;C:\Program Files\Resource Manager\bin;C:\Program Files\Microsoft Windows Performance Toolkit\;C:\Program Files\Java\jdk1.7.0_40\bin

PS C:\user> $env:path = $env:path + 'C:\Program Files (x86)\Microsoft Office\Office15'
PS C:\user>
PS C:\user> get-childitem env: | where-object { $_.Name -eq 'Path' } | format-table -wrap
Name Value
---- -----
Path %SystemRoot%\system32\WindowsPowerShell\v1.0\;C:\ProgramData\Oracle\Java\javapath;C:\Shared\Windows_Live;C:\oracle\11.2.0\client_1;C:\Windows\system32;C:\Windows;C:\Windows\Cryptosystems\NTRU_TCG_Software_Stack\bin\;C:\Program Files\NTRU_Cryptosystems\NTRU_TCG_Software_Stack\bin\;C:\Program Files\NTRU_TCG_Software_Stack\bin\;C:\Program Files (x86)\Common Files\Roxio Shared\DLLShared\;C:\Program Files (x86)\Common Files\Roxio Live\Shared;C:\Users\mgalea\NDI\ps_scripts;C:\Program Files (x86)\isc;C:\Program Files\Resource Manager\bin;C:\Program Files\Microsoft Windows Performance Toolkit\;C:\Program Files\Java\jdk1.7.0_40\bin;C:\Program Files (x86)\Microsoft Office\Office15
```

A new environmental variable may also be created in a similar manner. The following sequence of commands creates new environment variable called mysave containing the displayed value.

```
PS C:\Users> $env:mysave = 'd:\myprofile\mysave'
PS C:\Users> get-childitem env:mysave

Name          Value
----          -----
mysave        d:\myprofile\mysave
```

Once the current PowerShell session ends any environment variable modifications or new variables created disappear. PowerShell does not provide any cmdlets that persist new or changes to environment variables. However, cmdlets are available that allow modifying existing or creating new values in the Registry that correspond to environment variables.

The Variable Drive

Wednesday, January 16, 2013

9:01 AM

The variable: `drive` is the repository of all PowerShell variables whether automatic or user created. Unlike automatic variables which persist, user-created variables are removed from the variable: `drive` once a user exits a PowerShell session.

One of the significant advantages in virtualizing drives through the PSProviders, is that the common cmdlets may be used regardless of the type of drive.

Get-Childitem variable:

```
PS C:\Users> gci variable:  
Name                           Value  
----  
$                             clear  
?                             True  
^                             clear  
a                             {text, 255, ...}  
{}                            {}  
args                          High  
ConfirmPreference              SilentlyContinue  
ConsoleFileName                {You must specify an object for the Get-Me  
DebugPreference                Continue  
Error                         NormalView  
ErrorActionPreference           System.Management.Automation.Engineering  
ErrorView                      Context  
ExecutionContext                C:\Users\Tomato\Documents\WindowsPowerShell  
PWD                           Microsoft.PowerShell  
ShellId                        true  
StackTrace                     True  
v1                            1  
v2                            Hello  
v3                            3.5  
v4                            World  
VerbosePreference               SilentlyContinue  
WarningPreference               Continue  
whatIfPreference                False
```

Using the cmdlets already learned, we may search for specific variables as follows:

```
gci variable: | ? { $_.name -like 'v?'}
```

```
PS C:\Users>
PS C:\Users> gci variable: | ? { $_.name -like 'v?' }

Name          Value
----          ---
v1            1
v2            Hello
v3            3.5
v4            World
```

The following example provides the same results as the above but demonstrates the use of out-string and select-string. These two cmdlets are very helpful when searching for text strings as shown below.

```
gci variable: | out-string -stream | select-string '^v[0-9]'
```

```
PS C:\Users>
PS C:\Users> gci variable: | out-string -stream | select-string '^v[0-9]'

v1            1
v2            Hello
v3            3.5
v4            World
```

In the above, the output of get-childitem variable: is an array of objects of different types. The cmdlet out-string converts the default display output of the object into one string. The -stream parameter informs out-string not accumulate all the output of gci into one string but to pass individual strings down the pipeline. We use a regular expression with the select-string cmdlet to select all variables that begin with a v (case insensitive) followed by a digit. The regulars expression is interpreted as follows: ^ anchor to the beginning of the string, this means that the following character v must be at the beginning of the string, [0-9] the next character must match the range 0-9.

We may also indirectly alter variables as shown below. We create an array variable \$a and inspect the contents of the variable drive to see how the variable is represented.

```

PS C:\users>
PS C:\users> set-location variable:
PS Variable:>
PS Variable:> $a = 1,3,5,7,9,11
PS Variable:>
PS Variable:> get-childitem

Name          Value
----          -----
$              11
?              True
^              $a
a              {1, 3, 5, 7...}
ai             System.Management.Automation.PSVariable
args           {}
ConfirmPreference
General

```

The next sequence of steps show indirect manipulation of variables and also the notion of a object and the item (object) referenced by the object. We inspect create a variable \$ai as shown below. \$ai is Psvariable object referencing the variable \$a. We inspect the name and value properties and see that indeed \$ai is reference \$a. The next line shows that the type of \$a is array as we would expect. The statement \$ai.value = 'abc' changes the value of the reference object, which is \$a, to the string. This means that not only has the value of \$a changed but also its type. This is shown in the last sequence of steps.

```

PS Variable:\> PS Variable:\> $ai = get-item a
PS Variable:\> PS Variable:\> get-childitem a
Name                Value
----              -----
a                  {1, 3, 5, 7...}

PS Variable:\> get-childitem ai
Name                Value
----              -----
ai                 System.Management.Automation.PSVariable

PS Variable:\> $ai.name
a
PS Variable:\>
PS Variable:\> $ai.value
1
3
5
7
9
11
PS Variable:\> $a.gettype()
IsPublic IsSerial Name                                     BaseType
----- -----   ----
True     True    Object[]                                System.Array

PS Variable:\> $ai.value = 'abc'
PS Variable:\> PS Variable:\> get-childitem a
Name                Value
----              -----
a                  abc

PS Variable:\> $a.gettype()
IsPublic IsSerial Name                                     BaseType
----- -----   ----
True     True    String                                 System.Object

```

Managing Event Logs

Monday, March 17, 2014
1:18 PM

The Windows event logs contain a wealth of information about errors, status, and other hardware and software events. There are two cmdlets that may be used to query event logs: `get-eventlog` and `get-winevent`.

The cmdlet `get-eventlog` was traditionally used to query the Windows event logs. While it's easy to use, there are some drawbacks. The cmdlet `get-eventlog` has limited filtering capability so the log entries must be returned before they may be filtered with `where-object`. This cmdlet works only traditional online Windows logs but not with archived logs.

The cmdlet `get-winevent` has more robust filtering capability which provides more efficiency when query logs on remote hosts. It also works with archived logs so that log analysis may be performed more efficiently.

get-eventlog

This cmdlet returns an EventLogEntry object whose properties are based upon the log type. The columns in the eventlog are represented as properties in the log object returned by *get-eventlog*.

The syntax below lists the logs available to get-eventlog.

```
get-eventlog -list
```

Max(K)	Retain	OverflowAction	Entries	Log
20,480	0	OverwriteAsNeeded	4,500	Application
20,480	0	OverwriteAsNeeded	0	HardwareEvents
512	7	OverwriteOlder	0	Internet Explorer
20,480	0	OverwriteAsNeeded	0	Key Management Service
128	0	OverwriteAsNeeded	116	0Alerts
				Security
20,480	0	OverwriteAsNeeded	35,847	System
15,360	0	OverwriteAsNeeded	372	Windows PowerShell

The pipeline below may be used to discover the properties returned for a specific log. The log must always be specified otherwise the cmdelt will prompt for the log name.

```
get-eventlog -logname System | select-object -first 1 | gm
```

```
PS C:\Users>
PS C:\Users> get-eventlog -logname System | select-object -first 1 | gm

TypeName: System.Diagnostics.EventLogEntry#System/Windows-Kernel-General/1

Name          MemberType   Definition
----          -----      -----
Disposed      Event       System.EventHandler Disposed(System.Object, System.EventArgs)
CreateObjRef  Method      System.Runtime.Remoting.ObjRef CreateObjRef(type)
Dispose       Method      void Dispose(), void IDisposable.Dispose()
Equals        Method      bool Equals(System.Diagnostics.EventLogEntry)
GetHashCode   Method      int GetHashCode()
GetLifetimeService Method    System.Object GetLifetimeService()
GetObjectData  Method    void ISerializable.GetObjectData(System.Runtime.Serialization.SerializationInfo info, System.Runtime.Serialization.StreamingContext context)
GetType       Method      type GetType()
InitializeLifetimeService Method  System.Object InitializeLifetimeService()
ToString      Method      string ToString()
Category      Property    string Category {get;}
CategoryNumber Property    int16 CategoryNumber {get;}
Container     Property    System.ComponentModel.IContainer Container {get;}
Data          Property    byte[] Data {get;}
EntryType     Property    System.Diagnostics.EventLogEntryType EntryType
Index         Property    int Index {get;}
InstanceId    Property    long InstanceId {get;}
MachineName   Property    string MachineName {get;}
Message       Property    string Message {get;}
ReplacementStrings Property  string[] ReplacementStrings {get;}
Site          Property    System.ComponentModel.ISite Site {get;set;}
Source        Property    string Source {get;}
TimeGenerated  Property    datetime TimeGenerated {get;}
TimeWritten   Property    datetime TimeWritten {get;}
UserName      Property    string UserName {get;}
EventID       ScriptProperty System.Object EventID {get=$this.get_EventID()}


```

This example lists the most current 15 entries from the System log.

```
get-eventlog -logname system -newest 15
```

```
PS C:\Users>
PS C:\Users> get-eventlog -logname System -Newest 15
```

Index	Time	EntryType	Source	InstanceID	Message
3975	Mar 19 12:11	Information	Microsoft-Windows...	16	The description fo
3974	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3973	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3972	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3971	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3970	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3969	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3968	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3967	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3966	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3965	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3964	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3963	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3962	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo
3961	Mar 19 12:05	Information	Microsoft-Windows...	16	The description fo

We may also get specific types of events as shown below. In the first example, the syntax retrieves only warning and error events from the System log. The second example shows how we may retrieve logs from a remote hosts. Access to remote hosts is this and other scenarios is subject the credential verification.

```
get-eventlog system -entrytype error, warning
get-eventlog application -entrytype error, warning -computername
TI241-01
```

```
PS C:\Users>
PS C:\Users> get-eventlog system -entrytype error, warning
```

Index	Time	EntryType	Source	InstanceID	Message
3989	Mar 19 16:51	Warning	i8042prt	2147811345	The device sent an in
3988	Mar 19 16:51	Warning	Microsoft-Windows...	1014	Name resolution for t
3981	Mar 19 14:51	Warning	Microsoft-Windows...	1014	Name resolution for t
3924	Mar 19 10:44	Warning	Microsoft-Windows...	219	The driver \Driver\wu
3894	Mar 19 08:59	Error	Service Control M...	3221232495	The Update Orchestrat
3873	Mar 19 08:52	Error	Service Control M...	3221232495	The Update Orchestrat
3872	Mar 19 08:47	Error	DCOM	10010	The description for E
3859	Mar 18 16:29	Warning	Microsoft-Windows...	1014	Name resolution for t
3827	Mar 18 10:52	Error	Service Control M...	3221232495	The Update Orchestrat
3782	Mar 17 19:00	Warning	Microsoft-Windows...	1014	Name resolution for t
3775	Mar 17 16:53	Error	Service Control M...	3221232495	The Update Orchestrat
3771	Mar 17 16:09	Warning	Microsoft-Windows...	1014	Name resolution for t
3707	Mar 17 10:23	Error	Service Control M...	3221232495	The Update Orchestrat
3693	Mar 16 16:49	Error	Service Control M...	3221232495	The Update Orchestrat
3687	Mar 16 16:41	Error	Service Control M...	3221232495	The Update Orchestrat
3663	Mar 16 16:36	Error	Service Control M...	3221232477	The LoadUserProfile c
3632	Mar 16 16:35	Error	Service Control M...	3221232515	The Windows Update se
3631	Mar 16 16:35	Error	DCOM	10010	The description for E
3629	Mar 16 16:35	Error	DCOM	10010	The description for E

The -before and -after switches allow us to query events that occurred within a specific time range. In the example below we retrieve events that occurred with the one and half hour period between the two specified date-times.

```
get-eventlog system -after '09/19/2015 7:30 AM' -before '09/19/2015 8:30 AM'
```

Index	Time	EntryType	Source	InstanceID	Message
64180	Sep 19 08:07	Error	Service Control M...	3221232496	The Xbox Live Auth Manager service
64179	Sep 19 08:02	Information	Microsoft-Windows...	16	The description for Event ID '16
64178	Sep 19 07:46	Information	Microsoft-Windows...	20001	Driver Management concluded the
64177	Sep 19 07:46	Information	Microsoft-Windows...	98	The description for Event ID '98
64176	Sep 19 07:46	Information	WPDClassInstaller	1073831939	The description for Event ID '1073831939
64175	Sep 19 07:46	Information	WPDClassInstaller	1073831937	The description for Event ID '1073831937
64174	Sep 19 07:46	Information	WPDClassInstaller	1073831936	The description for Event ID '1073831936
64173	Sep 19 07:46	Information	Microsoft-Windows...	20003	Driver Management has concluded
64172	Sep 19 07:46	Information	Microsoft-Windows...	10100	The driver package installation
64171	Sep 19 07:46	Information	Microsoft-Windows...	10002	The UMDF service WpdFs (CLSID {10002}
64170	Sep 19 07:46	Information	Microsoft-Windows...	10000	A driver package which uses user
64169	Sep 19 07:44	Information	Virtual Disk Service	1107296260	Service stopped.
64168	Sep 19 07:39	Warning	disk	2147745843	An error was detected on device
64167	Sep 19 07:39	Warning	disk	2147745843	An error was detected on device
64166	Sep 19 07:39	Error	DCOM	10010	The description for Event ID '10010
64165	Sep 19 07:38	Warning	disk	2147745945	The IO operation at logical block
64164	Sep 19 07:37	Error	Service Control M...	3221232481	A timeout was reached (30000 mil
64163	Sep 19 07:36	Warning	disk	2147745945	The IO operation at logical block
64162	Sep 19 07:35	Information	Virtual Disk Service	1107296259	Service started.
64161	Sep 19 07:34	Warning	disk	2147745945	The IO operation at logical block
64160	Sep 19 07:32	Warning	disk	2147745945	The IO operation at logical block
64159	Sep 19 07:31	Error	Ntfs	55	A corruption was discovered in t
64158	Sep 19 07:31	Error	Microsoft-Windows...	98	The description for Event ID '98
64157	Sep 19 07:31	Error	Service Control M...	3221232483	A timeout (30000 milliseconds) w
64156	Sep 19 07:30	Information	Microsoft-Windows...	98	The description for Event ID '98

We use a bit of cleverness below to retrieve events that occurred within the past hour from the time when the cmdlet was executed. The subexpression (get-date) returns a [datetime] object representing the current date and time. We use the Addminutes method of a the object to return a date-time object referencing 60 minutes in the past.

```
get-eventlog Application -after (get-date).Addminutes(-60)
```

```

PS C:\users>
PS C:\users> get-eventlog Application -after (get-date).Addminutes(-60)

```

Index	Time	EntryType	Source	InstanceID	Message
4500	Sep 21 10:49	Information	VSS		8224 The VSS service is
4499	Sep 21 10:36	0	Software Protection	1073742727	The Software Protection
4498	Sep 21 10:36	Information	Software Protection	1073758208	Successfully scheduled
4497	Sep 21 10:36	Information	Software Protection	1073742827	The Software Protection
4496	Sep 21 10:36	0	Software Protection	1073742726	The Software Protection
4495	Sep 21 10:36	Information	Software Protection	1073742827	The Software Protection
4494	Sep 21 10:36	Information	Software Protection	1073742890	Initialization status
4493	Sep 21 10:36	Information	Software Protection	1073742724	The Software Protection
4492	Sep 21 10:36	Information	Outlook	1073741869	Outlook loaded the

```

get-eventlog application -after (get-date).Addminutes(-60) |
    select-object timewritten, eventid, message | sort-object -property
timewritten |
    format-table -autosize

```

```

PS C:\users>
PS C:\users> get-eventlog application -after (get-date).Addminutes(-60) |
>>> select-object timewritten, eventid, message | sort-object -property timewritten |
>>> format-table -autosize

```

TimeWritten	EventID	Message
9/21/2015 10:36:16 AM	1003	The Software Protection service has completed licensing...
9/21/2015 10:36:16 AM	45	Outlook loaded the following add-in(s):...
9/21/2015 10:36:16 AM	900	The Software Protection service is starting....
9/21/2015 10:36:16 AM	1066	Initialization status for service objects....
9/21/2015 10:36:16 AM	902	The Software Protection service has started....
9/21/2015 10:36:16 AM	1003	The Software Protection service has completed licensing...
9/21/2015 10:36:47 AM	903	The Software Protection service has stopped....
9/21/2015 10:36:47 AM	16384	Successfully scheduled Software Protection service for
9/21/2015 10:49:31 AM	8224	The VSS service is shutting down due to idle timeout.

One of the issues apparent when retrieving event logs from remote host is that response time would be problematic since the full or partial event log must be transferred to the local host. To make such transfers more efficient, qualifying the entries to retrieve is required. The next example illustrates this principle in that it retrieves entries for two consecutive days.

```

get-eventlog system -after ([datetime] '09/18/2015 12:00 AM') ` 
    -before ([datetime] '09/19/2015 11:59 PM') |
    format-table -autosize

```

Index	Time	EntryType	Source	InstanceID	Message
64216	Sep 19 22:41	Information	Microsoft-Windows-Kernel-General	1	The c
64215	Sep 19 22:41	Information	Microsoft-Windows-Kernel-General	1	The c
64214	Sep 19 22:41	Information	Microsoft-Windows-Time-Service	35	The t
64213	Sep 19 22:41	Information	Microsoft-Windows-Kernel-General	1	The c
64212	Sep 19 22:41	Information	Microsoft-Windows-Time-Service	37	The t
64211	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64210	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64209	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64208	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64207	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64206	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64205	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64204	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64203	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64202	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64201	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64200	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64199	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64198	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64197	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64196	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c
64195	Sep 19 21:29	Information	Microsoft-Windows-Kernel-General	16	The c

To further reduce processing, we may qualify the query by specifying the sources of the events.

```
get-eventlog system -after ([datetime] '09/18/2015 12:00 AM') ` 
    -before ([datetime] '09/19/2015 11:59 PM') ` 
    -source DCOM, disk | 
format-table -autosize
```

Index	Time	EntryType	Source	InstanceID	Message
64168	Sep 19 07:39	Warning	disk	2147745843	An error was detected on device \Device\Harddisk7\
64167	Sep 19 07:39	Warning	disk	2147745843	An error was detected on device \Device\Harddisk7\
64166	Sep 19 07:39	Error	DCOM	10010	The description for Event ID '10010' in Source 'DC'
64165	Sep 19 07:38	Warning	disk	2147745945	The IO operation at logical block address 0x68f469
64163	Sep 19 07:36	Warning	disk	2147745945	The IO operation at logical block address 0x68f469
64161	Sep 19 07:34	Warning	disk	2147745945	The IO operation at logical block address 0x68f469
64160	Sep 19 07:32	Warning	disk	2147745945	The IO operation at logical block address 0x68f469
64136	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64135	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64134	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64133	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64132	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64131	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64130	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64129	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64128	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64127	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64126	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64125	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64124	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64123	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64122	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64121	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'
64120	Sep 19 07:25	Error	DCOM	10016	The description for Event ID '10016' in Source 'DC'

If we are interested in only a certain type of entry, we may further reduce processing by selecting only specific entry types.

```

PS C:\users>
PS C:\users> get-eventlog system -after ([datetime] '09/18/2015 12:00 AM') ` 
>>>                                -before ([datetime] '09/19/2015 11:59 PM')
>>>                                -source DCOM, disk
>>>                                -entrytype warning |
>>> format-table -autosize

Index Time          EntryType Source InstanceID Message
----- --          ----- --     ----- --      -----
64168 Sep 19 07:39 Warning disk   2147745843 An error was detected on device \Device\Harddi
64167 Sep 19 07:39 Warning disk   2147745843 An error was detected on device \Device\Harddi
64165 Sep 19 07:38 Warning disk   2147745945 The IO operation at logical block address 0x68
64163 Sep 19 07:36 Warning disk   2147745945 The IO operation at logical block address 0x68
64161 Sep 19 07:34 Warning disk   2147745945 The IO operation at logical block address 0x68
64160 Sep 19 07:32 Warning disk   2147745945 The IO operation at logical block address 0x68
63852 Sep 18 22:19 Warning disk   2147745945 The IO operation at logical block address 0x28
63850 Sep 18 22:17 Warning disk   2147745945 The IO operation at logical block address 0x28
63848 Sep 18 22:15 Warning disk   2147745945 The IO operation at logical block address 0x28
63847 Sep 18 22:13 Warning disk   2147745945 The IO operation at logical block address 0x28
63846 Sep 18 22:12 Warning disk   2147745945 The IO operation at logical block address 0x28

```

This example taken from the Microsoft TechNet site provides a summary of the type of events present in a log.

```

get-eventlog system -newest 500 | 
    group-object -property source -noelement | 
    sort-object -property count -descending | 
    format-table -auto

```

```

PS C:\users>
PS C:\users> get-eventlog system -newest 500 |
>>> group-object -property source -noelement |
>>> sort-object -property count -descending |
>>> format-table -auto

Count Name
-----
107 DCOM
 68 Microsoft-Windows-Kernel-General
 38 Service Control Manager
 36 Microsoft-Windows-Kernel-Processor-Power
 30 Microsoft-Windows-WindowsUpdateClient
 30 Microsoft-Windows-DriverFrameworks-UserMode
 29 Microsoft-Windows-Hyper-V-VmSwitch
 22 Microsoft-Windows-Ntfs
 21 Microsoft-Windows-FilterManager
 17 disk
 16 EventLog
 15 Microsoft-Windows-Kernel-Boot
   7 eliexpress
   6 Microsoft-Windows-Hyper-V-Hypervisor
   5 Microsoft-Windows-Dhcp-Client
   5 WPDClassInstaller
   5 Microsoft-Windows-DHCPv6-Client
   4 Microsoft-Windows-Kernel-Power
   4 RasMan
   4 Virtual Disk Service
   4 Microsoft-Windows-UserPnp
   4 Ntfs
   3 Microsoft-Windows-DNS-Client
   3 Microsoft-Windows-Kernel-PnP
   3 Microsoft-Windows-Wininit
   3 Microsoft-Windows-Winlogon
   3 Microsoft-Windows-Time-Service
   2 User32
   2 volsnap
   2 volmgr
   1 Application Popup
   1 Tcpip

```

A variation of the above pipeline provides a summary by entry type.

```

get-eventlog system -newest 500 |
  group-object -property entrytype -noelement |
  sort-object -property count -descending |
  format-table -auto

```

```
PS C:\users>
PS C:\users> get-eventlog system -newest 500 |
>>> group-object -property entrytype -noelement |
>>> sort-object -property count -descending |
>>> format-table -auto

Count Name
---- --
298 Information
155 Error
24 0
23 Warning
```

get-winevent

What is most notable is that get-winevent returns a different type of object compared to get-eventlog. The object type is EventLogRecord and provides different properties and methods. The other difference between these cmdlets is that get-winevent, while more powerful, is more difficult to use since some of the query parameters, like -EntryType, available to get-eventlog are not present in get-winevent. The cmdlet get-winevent relies on a query hash table to provide the same functionality.

```
get-winevent -logname System | select-object -first 1 | get-member
```

TypeName: System.Diagnostics.Eventing.Reader.EventLogRecord		
Name	MemberType	Definition
Dispose	Method	void Dispose(), void IDisposable.Dispose()
Equals	Method	bool Equals(System.Object obj)
FormatDescription	Method	string FormatDescription(), string FormatDescription(System.Collections.Generic.IList<System.Object> collection)
GetHashCode	Method	int GetHashCode()
GetPropertyValues	Method	System.Collections.Generic.IList<System.Object> GetPropertyValues(System.Collections.Generic.IDictionary<string, object> properties)
GetType	Method	type GetType()
ToString	Method	string ToString()
ToXml	Method	string ToXml()
Message	NoteProperty	System.String Message=The start type of the Background Intelligent Transfer System.
ActivityId	Property	System.Nullable<guid> ActivityId {get;}
Bookmark	Property	System.Diagnostics.Eventing.Reader.EventBookmark Bookmark {get;}
ContainerLog	Property	string ContainerLog {get;}
Id	Property	int Id {get;}
Keywords	Property	System.Nullable<long> Keywords {get;}
KeywordsDisplayNames	Property	System.Collections.Generic.IEnumerable<string> KeywordsDisplayNames {get;}
Level	Property	System.Nullable<byte> Level {get;}
LevelDisplayName	Property	string LevelDisplayName {get;}
LogName	Property	string LogName {get;}
MachineName	Property	string MachineName {get;}
MatchedQueryIds	Property	System.Collections.Generic.IEnumerable<int> MatchedQueryIds {get;}
Opcode	Property	System.Nullable<int16> Opcode {get;}
OpcodeDisplayName	Property	string OpcodeDisplayName {get;}
ProcessId	Property	System.Nullable<int> ProcessId {get;}
Properties	Property	System.Collections.Generic.IList<System.Diagnostics.Eventing.Reader.EventProperty> Properties {get;}
ProviderId	Property	System.Nullable<guid> ProviderId {get;}
ProviderName	Property	string ProviderName {get;}
Qualifiers	Property	System.Nullable<int> Qualifiers {get;}
RecordId	Property	System.Nullable<long> RecordId {get;}
RelatedActivityId	Property	System.Nullable<guid> RelatedActivityId {get;}
Task	Property	System.Nullable<int> Task {get;}
TaskDisplayName	Property	string TaskDisplayName {get;}
ThreadId	Property	System.Nullable<int> ThreadId {get;}
TimeCreated	Property	System.Nullable<datetime> TimeCreated {get;}
UserId	Property	System.Security.Principal.SecurityIdentifier UserId {get;}
Version	Property	System.Nullable<byte> Version {get;}

As shown below, what is most apparent is the difference in the number of logs available to get-winevent relative to get-eventlog.

```
get-winevent -listlog * -erroraction SilentlyContinue
```

```

PS C:\Users>
PS C:\Users> get-winevent -ListLog * -erroraction SilentlyContinue

```

LogMode	MaximumSizeInBytes	RecordCount	LogName
Circular	20971520	8493	Application
Circular	20971520	0	HardwareEvents
Circular	1052672	0	Internet Explorer
Circular	20971520	0	Key Management Service
Circular	1052672	145	OAlerts
Circular	20971520	7036	System
Circular	15728640	519	Windows PowerShell
Circular	20971520		ForwardedEvents
Circular	1052672	0	Microsoft-Windows-All-User-Install-Agent/Admin
Circular	1052672	5	Microsoft-Windows-AppHost/Admin
Circular	1052672	0	Microsoft-Windows-AppID/Operational
Circular	1052672	0	Microsoft-Windows-ApplicabilityEngine/Operational
Circular	1052672	0	Microsoft-Windows-Application Server-Applications/Admin
Circular	1052672	0	Microsoft-Windows-Application Server-Applications/Operational
Circular	1052672	51	Microsoft-Windows-Application-Experience/Program-Compatibility
Circular	1052672	0	Microsoft-Windows-Application-Experience/Program-Compatibility
Circular	1052672	119	Microsoft-Windows-Application-Experience/Program-Inventory
Circular	1052672	1543	Microsoft-Windows-Application-Experience/Program-Telemetry
Circular	1052672	0	Microsoft-Windows-Application-Experience/Steps-Recorder
Circular	1052672	1152	Microsoft-Windows-ApplicationResourceManagementSystem/Operational
Circular	1052672	0	Microsoft-Windows-AppLocker/EXE and DLL
Circular	1052672	0	Microsoft-Windows-AppLocker/MSI and Script
Circular	1052672	0	Microsoft-Windows-AppLocker/Packaged app-Deployment
Circular	1052672	0	Microsoft-Windows-AppLocker/Packaged app-Execution
Circular	1052672	1338	Microsoft-Windows-AppModel-Runtime/Admin
Circular	5242880	159	Microsoft-Windows-AppReadiness/Admin
Circular	5242880	1249	Microsoft-Windows-AppReadiness/Operational
Circular	1052672	239	Microsoft-Windows-AppXDeployment/Operational
Circular	5242880	3440	Microsoft-Windows-AppXDeploymentServer/Operational
Circular	1052672	274	Microsoft-Windows-AppxPackaging/Operational
Circular	1052672	0	Microsoft-Windows-AssignedAccess/Admin
Circular	1052672		Microsoft-Windows-AssignedAccess/Operational
Circular	1052672	0	Microsoft-Windows-Audio/CaptureMonitor
Circular	1052672	2425	Microsoft-Windows-Audio/GlitchDetection
Circular	1052672		Microsoft-Windows-Audio/Informational
Circular	1052672	0	Microsoft-Windows-Audio/Operational
Circular	1052672	2255	Microsoft-Windows-Audio/PlaybackManager
Circular	1052672	194	Microsoft-Windows-Authentication User Interface/Operational
Circular	1052672		Microsoft-Windows-Authentication/AuthenticationPolicyFailure
Circular	1052672		Microsoft-Windows-Authentication/ProtectedUser-Client

This cmdlet also allows the user to query information about specific logs.

```
get-winevent -listlog "Windows PowerShell", "Setup"
```

```

PS C:\Users>
PS C:\Users> get-winevent -listlog "Windows PowerShell", "Setup"

```

LogMode	MaximumSizeInBytes	RecordCount	LogName
Circular	15728640	519	Windows PowerShell
Circular	1052672	2052	Setup

The example below shows how to retrieve today's entries from the system log. Using query filters requires the parameter `-FilterHashTable` and a corresponding hash table with the selection criteria.

```
Get-WinEvent -FilterHashtable @{logname='application';  
StartTime=(get-date).date}
```

```
PS C:\Users> Get-WinEvent -FilterHashtable @{logname='application'; StartTime=(get-date).date}

ProviderName: Microsoft-Windows-LocationProvider
TimeCreated                Id LevelDisplayName Message
-----                -- -- -- -- 
3/22/2015 2:24:07 PM      2003 Information     The Windows Location Provider has successfully shutdown...
3/22/2015 2:24:07 PM      2001 Information     The Windows Location Provider has successfully started...
3/22/2015 2:24:01 PM      2003 Information     The Windows Location Provider has successfully shutdown...
3/22/2015 2:24:01 PM      2001 Information     The Windows Location Provider has successfully started...

ProviderName: LocationNotifications
TimeCreated                Id LevelDisplayName Message
-----                -- -- -- -- 
3/22/2015 2:24:01 PM      1   Information     A program accessed information from a location sensor

ProviderName: Microsoft-Windows-Security-SPP
TimeCreated                Id LevelDisplayName Message
-----                -- -- -- -- 
3/22/2015 2:11:25 PM      903 Information     The Software Protection service has stopped.....
3/22/2015 2:11:25 PM      16384 Information  Successfully scheduled Software Protection service for ...
3/22/2015 2:10:54 PM      1003 Information    The Software Protection service has completed licensin...
3/22/2015 2:10:54 PM      902 Information     The Software Protection service has started.....
3/22/2015 2:10:54 PM      1003 Information    The Software Protection service has completed licensin...
3/22/2015 2:10:54 PM      1066 Information   Initialization status for service objects.....
3/22/2015 2:10:54 PM      900 Information     The Software Protection service is starting.....
3/22/2015 2:07:28 PM      903 Information     The Software Protection service has stopped.....
3/22/2015 2:07:28 PM      16384 Information  Successfully scheduled Software Protection service for ...
```

Retrieving errors or warnings using get-winevent requires knowing the various log levels: undefined=0, critical=1, error=2, warning=3, information=4, verbose=5. Knowing the above, retrieving errors from the system log that occurred today requires the syntax below.

```
Get-WinEvent -FilterHashtable @{logname='system'; level=2; StartTime=(get-date).date}
```

```

PS C:\Users> Get-WinEvent -FilterHashtable @{logname='system'; Level=2; StartTime=(get-date).date}

    ProviderName: Service Control Manager
TimeCreated                Id LevelDisplayName Message
---- -- -- -- -- -- -- -- 
3/22/2015 12:29:29 PM      7031 Error          The HauppaugeTVServer service terminated unexped

PS C:\Users> Get-WinEvent -ComputerName localhost -FilterHashtable @{logname='system'; level=2; StartT
ime=(get-date).date}

    ProviderName: Service Control Manager
TimeCreated                Id LevelDisplayName Message
---- -- -- -- -- -- -- -- 
3/22/2015 12:29:29 PM      7031 Error          The HauppaugeTVServer service terminated unexped

```

We may also add a -ComputerName parameter to retrieve remote log information.

```

Get-WinEvent -ComputerName TI241-01 `

           -FilterHashtable @{logname='system'; level=2;
StartTime=(get-date).date}

```

In the example below we retrieve system log events that occurred between 12:00 PM and 1:30 PM today.

```

Get-WinEvent -FilterHashtable @{logname='system';
                           StartTime=((get-
date).date).AddHours(12);
                           EndTime=((get-
date).date).AddHours(13.5)}

```

```

PS C:\Users> Get-WinEvent -FilterHashtable @{logname='system';
>>                                         StartTime=((get-date).date).AddHours(12);
>>                                         EndTime=((get-date).date).AddHours(13.5)}

    ProviderName: Service Control Manager
TimeCreated                Id LevelDisplayName Message
---- -- -- -- -- -- -- -- 
3/22/2015 12:52:31 PM      7040 Information   The start type of the Background Intelligent
3/22/2015 12:29:29 PM      7031 Error        The HauppaugeTVServer service terminated unexped

    ProviderName: EventLog
TimeCreated                Id LevelDisplayName Message
---- -- -- -- -- -- -- -- 
3/22/2015 12:00:00 PM      6013 Information   The system uptime is 970973 seconds.

```

In the above cmdlet execution, we use some cleverness to specify the start time to begin selecting events and the end time to stop selecting events. The cmdlet get-date retrieves a date-time object. The date property of the returned object, (get-date).date , returns a date-time object representing 12:00 AM of the current date. We then use the AddHours method to add 12 hours to the date-time object which results in a date-object representing 12:00 PM of the current date. We use the same expression and add 13.5 hours to create a date-time object representing 1:30 PM of the current date.

```
PS C:\Users>
PS C:\Users> get-date
Sunday, March 22, 2015 6:50:30 PM

PS C:\Users> (get-date).date
Sunday, March 22, 2015 12:00:00 AM

PS C:\Users> ((get-date).date).AddHours(12)
Sunday, March 22, 2015 12:00:00 PM

PS C:\Users> ((get-date).date).AddHours(13.5)
Sunday, March 22, 2015 1:30:00 PM
```

Events that occurred within the past hour from when the cmdlet is executed may be retrieved using what we learned in the above. We use the subexpression (get-date).AddMinutes(-60) to instantiate a date-time object representing one hour from the time of execution.

```
Get-WinEvent -FilterHashtable @{logname='system'; StartTime=(get-date).AddMinutes(-60)}
```

```
PS C:\Users> Get-WinEvent -FilterHashtable @{Logname='system'; StartTime=(get-date).AddMinutes(-60)}
```

TimeCreated	Id	LevelDisplayName	Message
3/22/2015 6:17:36 PM	7040	Information	The start type of the Background Intelligent Transfer

```
Get-WinEvent -FilterHashtable @{logname='system';  
                                StartTime=(get-date).AddMinutes(-  
60)} |  
    select-object timewritten, eventid, message | sort-object  
timewritten |  
    format-table -autosize
```

```
PS C:\Users>
PS C:\Users> get-eventlog system -after (get-date).Addminutes(-60) |
>>         select-object timewritten, eventid, message | sort-object ti
>>     format-table -autosize
>>

TimeWritten          EventID Message
-----  ----- 
3/19/2015 2:19:49 PM      42 The description for Event ID '42' in Source 'M
3/19/2015 2:51:46 PM      1 The description for Event ID '1' in Source 'Mi
3/19/2015 2:51:48 PM    131 The description for Event ID '131' in Source '
3/19/2015 2:51:51 PM      1 The system has returned from a low power state
3/19/2015 2:51:52 PM     11 Broadcom NetXtreme 57xx Gigabit Controller: Ne
3/19/2015 2:51:55 PM   1014 Name resolution for the name wpad timed out af
```

We can select events generated by different providers which in get-eventlog jargon are called sources. The example below displays an events generate by the NTFS file system and the volume manager generated within the past week. Note, 168 hours is the number of hours in seven days.

```
Get-WinEvent -FilterHashtable ` 
    @{logname='system';ProviderName='volmgr','Microsoft-Windows-Ntfs';` 
    StartTime=(get-date).AddHours(-168) }
```

```
PS C:\Users>
PS C:\Users> Get-WinEvent -FilterHashtable @{$logname='system'; ProviderName='volmgr','Microsoft-Windows-Ntfs'; StartTime=(get-date).AddHours(-168) }

ProviderName: Microsoft-Windows-Ntfs
TimeCreated           Id LevelDisplayName Message
-- -- -- -- -- -- -- --
3/20/2015 9:27:50 PM 98 Information    Volume ?? (\Device\HarddiskVolumeShadowCopy15) is healthy.
3/20/2015 9:27:49 PM 98 Information    Volume Recovery (\Device\HarddiskVolumeShadowCopy14) is healthy.
3/20/2015 7:09:39 AM 98 Information    Volume ?? (\Device\HarddiskVolumeShadowCopy13) is healthy.
3/19/2015 6:11:24 AM 98 Information    Volume M: (\Device\HarddiskVolume31) is healthy. No action
```

Managing Processes

Tuesday, April 9, 2013
1:13 PM

Monitoring Processes:

List all processes currently executing.

```
get-process
```

```
PS C:\Users>
PS C:\Users> get-process
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
81	7	1164	948	44		1648	armsvc
224	13	9896	10308	72	2,511.00	7004	audiogd
141	10	2288	6552	54		7164	CaptureDLNA
137	10	1996	6272	53		9784	CaptureDLNA
138	10	2000	6272	53		10228	CaptureDLNA
234	17	15220	12944	84		15476	CaptureGenUSB
214	23	45784	56412	243	0.70	1784	chrome
179	22	43132	49108	232	1.22	2556	chrome
229	32	65608	58752	243	22.03	3024	chrome
2237	94	149188	199836	480	544.75	4896	chrome
159	20	28440	30064	219	0.73	6528	chrome
165	20	34568	36776	223	3.63	6900	chrome
157	18	22788	23752	203	0.53	7152	chrome
158	19	26992	29884	213	1.83	7300	chrome
159	19	24344	25444	204	0.69	7648	chrome
175	29	101688	101808	299	50.09	10208	chrome
175	25	92576	93560	284	2.11	10608	chrome
168	20	35028	37624	226	7.61	10692	chrome
210	25	25016	43056	226	0.39	11016	chrome
397	38	46564	68236	307	168.17	11760	chrome
220	36	140592	150228	347	169.05	11996	chrome
179	23	54492	60844	248	1.67	12260	chrome
173	21	37728	40368	225	0.69	12876	chrome

We may list specific processes using the full process name or a partial name with wildcards.

```
get-process -name explorer, powershell
get-process exp*, power*
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
3029	136	112748	166572	773	285.27	15616	explorer
634	29	160708	169888	665	22.92	17720	powershell
PS C:\Users> get-process exp*, power*							
Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
3050	137	122648	175752	783	285.38	15616	explorer
429	29	155760	164960	665	22.92	17720	powershell
573	56	176856	201672	895	234.63	8676	powershell_ise

This cmdlet may also be executed on remote hosts.

```
get-process -ComputerName Server01, Server01 |  
Format-Table -Property ID, ProcessName, MachineName
```

Since most malware has no corporate identification, a useful query is to list the process names which have no company description.

```
get-process | Where { $_.Company -eq $null } | format-table  
Processname
```

```
PS C:\Users>  
PS C:\Users> Get-Process | Where { $_.Company -eq $null } | format-table Processname  
ProcessName  
-----  
armsvc  
audiogd  
CaptureDLNA  
CaptureDLNA  
CaptureDLNA  
CaptureGenUSB  
conhost  
conhost  
csrss  
csrss  
CtHdaSvc  
FileMon
```

We can also list process that started within past two hours.

```
get-process | ? { $_.StartTime -gt (get-date).AddMinutes(-120) } |  
format-table -property Name, Company -autosize
```

```
PS C:\Users>  
PS C:\Users> get-process | ? { $_.StartTime -gt (Get-Date).AddMinutes(-120) } |  
>>         ft -property Name, Company -autosize  
>>  
Name          Company  
----          -----  
chrome        Google Inc.  
chrome        Google Inc.  
conhost       Microsoft Corporation  
ONENOTE       Microsoft Corporation  
powershell    Microsoft Corporation  
powershell_ise Microsoft Corporation  
SearchProtocolHost Microsoft Corporation  
SnippingTool  Microsoft Corporation
```

Starting Processes:

The cmdlet start-process is used to start a process. However, start-process does not return a process object. An alternative is to use Start method of the .Net class [System.Diagnostics.Process] to start the process. This method does return a process object so the process may be managed.

```
$p = Start-Process notepad.exe  
$p | gm  
  
$p = [System.Diagnostics.Process]::Start('notepad.exe')  
$p | gm
```

```
PS C:\Users>  
PS C:\Users> $p = Start-Process notepad.exe  
PS C:\Users> $p | gm  
gm : You must specify an object for the Get-Member cmdlet.  
At line:1 char:6  
+ $p | gm  
+  
+     + CategoryInfo          : CloseError: (:) [Get-Member], InvalidOperationException  
+     + FullyQualifiedErrorId : NoObjectInGetMember,Microsoft.PowerShell.Commands.Ge  
  
PS C:\Users>  
PS C:\Users> $p = [System.Diagnostics.Process]::Start('notepad.exe')  
PS C:\Users>  
PS C:\Users> $p | gm
```

Type Name: System.Diagnostics.Process

Name	MemberType	Definition
Handles	AliasProperty	Handles = HandleCount
Name	AliasProperty	Name = ProcessName
NPM	AliasProperty	NPM = NonpagedSystemMemorySize
PM	AliasProperty	PM = PagedMemorySize
VM	AliasProperty	VM = VirtualMemorySize
WS	AliasProperty	WS = WorkingSet
Disposed	Event	System.EventHandler Disposed(System.Object)
ErrorDataReceived	Event	System.Diagnostics.DataReceivedEventHandler
Exited	Event	System.EventHandler Exited(System.Object)
OutputDataReceived	Event	System.Diagnostics.DataReceivedEventHandler
BeginErrorReadLine	Method	void BeginErrorReadLine()
BeginOutputReadLine	Method	void BeginOutputReadLine()
CancelErrorRead	Method	void CancelErrorRead()
CancelOutputRead	Method	void CancelOutputRead()

The above works because notepad.exe is located in c:\windows\system32 which is found in the PATH environment variable. If the location of the executable is not in the PATH, the full qualified file name must be specified.

```
# this doesn't work
$pexe = 'excel.exe'
start-process $pexe
```

The above doesn't work because the directory where excel.exe is located is not in the PATH. So, one option is to use a fully qualified file name.

```
$pexe= 'C:\Program Files (x86)\Microsoft Office\Office15\excel.exe'
start-process $pexe

$p = [System.Diagnostics.Process]::Start($pexe)
```

An alternative option is to alter the PATH variable to include the Office path as below:

```
$env:path = $env:path + ';'C:\Program Files (x86)\Microsoft
Office\Office15'
$pexe = 'excel.exe'
start-process $pexe
```

In the above the semicolon preceding the path is required to separate this path from the preceding path. The change to the path variable does not persist over multiple sessions; so once the session ends the path variable reverts back to the original setting.

Stopping Processes:

A number of different methods for stopping processes are available. The cmdlet stop-process is available but has some disadvantages.

```
start-process notepad
stop-process notepad
```

The issue is that when multiple identically named processes are active, stopping a process using the name stops all processes with that name.

```
start-process notepad
start-process notepad
stop-process -name notepad
```

The stop-process execution above is identical to doing the following.

```
stop-process -name notepad*
```

A solution to this problem requires distinguishing between the processes using the process id. However, since start-process does not return a process object. We need to get the process object and save its id after it was started.

```
start-process notepad
get-process notepad
$pid1 = (get-process notepad).id
start-process notepad
$pid2 = (get-process notepad | ? { $_.id -ne $pid1 }).id
```

After the above completes, we have the process id of the first instance of notepad in \$pid1 and the process id of the second instance of notepad in \$pid2. Now we can selectively stop the process using the process ids as below.

```
stop-process -id $pid1
stop-process -id $pid2
```

There are still some issues that may arise from using the above steps. What happens if another instance of notepad is started between the execution of the first instance and the second instance in the above script. In this case, \$pid2 is no longer a singleton (scalar) but an array. Stopping the process using \$pid2 would stop the remaining instances of notepad which may not be desired.

The above issues arise simply because start-process does not return a process object. When writing scripts to manage processes, using the .Net class [System.Diagnostics.Process] is the better option. The script below is a better implementation of the above script.

```
$pid1 = ([System.Diagnostics.Process]::Start('notepad.exe')).id
$pid2 = ([System.Diagnostics.Process]::Start('notepad.exe')).id
start-sleep -seconds 5
stop-process -id $pid1
```

```
stop-process -id $pid2
```

We can even improve upon the above by saving the process objects for the two instances of notepad and use methods to stop them.

```
$p1 = [System.Diagnostics.Process]::Start('notepad.exe')
$p2 = [System.Diagnostics.Process]::Start('notepad.exe')
start-sleep -seconds 5
$p1.kill()
$p2.kill()
```

Waiting for process to complete:

The wait-process cmdlet suspends the execution of the script and waits until the process ends before the script can continue to execute.

```
start-process notepad
wait-process -name notepad
write-host 'This executes after notepad ends'
```

Managing Services

Tuesday, April 9, 2013
3:24 PM

Managing Services:

The get-service cmdlet returns service object(s) that may be used to manage services. Running get-service as shown below returns an array of service objects.

```
$s = get-service
$s = get-service -computername server01
```

Services have name and a display name. The name is a short internal name. The display name is a more descriptive name that appears in the Computer

Management -> Services window. Running get-service using either construct results in one service object returned.

```
get-service -name wsearch  
get-service -displayname 'Windows Search'
```

```
PS C:\users>  
PS C:\users> get-service -name wsearch  


| Status  | Name    | DisplayName    |
|---------|---------|----------------|
| Running | wsearch | Windows Search |

  
PS C:\users> get-service -displayname 'Windows Search'  


| Status  | Name    | DisplayName    |
|---------|---------|----------------|
| Running | WSearch | Windows Search |


```

We may list services that are dependent on other services or the dependencies of services. For example to list the services required by Windows Search the commands below may be executed.

```
get-service -displayname 'Windows Search' -RequiredServices  
get-service -name wsearch -RequiredServices
```

```
PS C:\users>  
PS C:\users> get-service -displayname 'Windows Search' -RequiredServices  


| Status  | Name  | DisplayName                 |
|---------|-------|-----------------------------|
| Running | RPCSS | Remote Procedure Call (RPC) |

  
PS C:\users> get-service -name wsearch -RequiredServices  


| Status  | Name  | DisplayName                 |
|---------|-------|-----------------------------|
| Running | RPCSS | Remote Procedure Call (RPC) |


```

We may also list the services dependent on a particular service. The example below lists the services dependent upon the lanmanworkstation service.

PS C:\users> get-service -name Lanmanworkstation -DependentServices			
Status	Name	DisplayName	
Running	VMwareHostd	VMware Workstation Server	
Stopped	SessionEnv	Remote Desktop Configuration	
Stopped	NetLogon	Netlogon	
Running	Browser	Computer Browser	

The pipeline below lists all services that have required services or dependencies.

```
get-Service | where {$_.RequiredServices -or $_.DependentServices} |
Format-Table -Property Status, Name, RequiredServices,
DependentServices -auto
```

PS C:\users> Get-Service where {\$_.RequiredServices -or \$_.DependentServices} Format-Table -Property Status, Name, RequiredServices, DependentServices -auto			
Status	Name	RequiredServices	DependentServices
Stopped	AppIDSvc	[RpcSs, CryptSvc, AppID]	{}
Running	Appinfo	[RpcSs, ProfSvc]	{}
Stopped	AppxSvc	[rpcss]	{}
Running	AudioEndpointBuilder	[]	{Audiosrv}
Running	Audiosrv	[MMCSS, RpcSs, AudioEndpointBuilder]	{}
Stopped	AxInstSV	[rpcss]	{}
Running	BFE	[RpcSs, WfpLwfs]	{WdNisSvc, WdNisDrv, SharedAcc}
Running	BITS	[RpcSs, EventSystem]	{}
Running	BrokerInfrastructure	[DcomLaunch, RpcSs, RpcEptMapper]	{}
Running	Browser	[LanmanServer, LanmanWorkstation]	{}
Stopped	BthHFSrv	[bthserv]	{BthHFSrv}
Stopped	bthserv	[]	{}
Stopped	CertPropSvc	[RpcSs]	{}
Stopped	COMSysApp	[EventSystem, SENS, RpcSs]	{}
Running	CryptSvc	[RpcSs]	{AppIDSvc}
Running	CscService	[RpcSs]	{}
Running	DcomLaunch	[]	{WwanSvc, wuauserv, WSService, }
Stopped	defragsvc	[RPCSS]	{}
Running	Dhcp	[Afd, NSI, Tdx]	{NcaSvc, iphlpsvc, WinHttpAuto}
Running	DnsCache	[nsi, Tdx]	{NcaSvc}
Stopped	dot3svc	[RpcSs, Eaphost, Ndisuio]	{}
Stopped	DsmSvc	[RpcSs, HTTP]	{}
Stopped	EventLog	[RPCSS, Kernel]	{}

Since we're generally interested in running services, we may alter the pipeline above as below.

```
get-service | where ( $_.status -eq 'Running' ) |
where {$_.RequiredServices -or $_.DependentServices} |
Format-Table -Property Status, Name, RequiredServices,
DependentServices -auto
```

The following example shows the use other cmdlets available to stop and start services. We first attempt to stop the Windows Search service which fails since

the Windows Media Player Networking Sharing service is dependent on Search. In addition to these cmdlets, other cmdlets exists to suspend and restart services.

```
stop-service -name wsearch -verbose  
get-service -name wsearch -dependentservices  
stop-service -name WMPNetworkSvc  
stop-service -name wsearch -verbose
```

```
PS C:\users> stop-service -name wsearch -verbose  
VERBOSE: Performing the operation "Stop-Service" on target "Windows Search (wsearch)".  
stop-service : Cannot stop service 'Windows Search (wsearch)' because it has dependent services stopped if the Force flag is set.  
At line:1 char:1  
+ stop-service -name wsearch -verbose  
+ ~~~~~~  
    + CategoryInfo          : InvalidOperation: (System.ServiceProcess.ServiceController:Service), ServiceCommandException  
    + FullyQualifiedErrorId : ServiceHasDependentServices,Microsoft.PowerShell.Commands.StopService  
  
PS C:\users> get-service -name wsearch -dependentservices  


| Status  | Name          | DisplayName                            |
|---------|---------------|----------------------------------------|
| Stopped | workfoldersvc | Work Folders                           |
| Running | WMPNetworkSvc | Windows Media Player Network Sharin... |

  
PS C:\users> stop-service -name WMPNetworkSvc  
PS C:\users> stop-service -name wsearch -verbose  
VERBOSE: Performing the operation "Stop-Service" on target "Windows Search (wsearch)".  
PS C:\users>
```

We start the services stopped in the previous step.

```
start-service -name wsearch -verbose  
start-service -name WMPNetworkSvc -verbose
```

```
PS C:\users>  
PS C:\users> start-service -name wsearch -verbose  
VERBOSE: Performing the operation "Start-Service" on target "Windows Search (wsearch)".  
PS C:\users> start-service -name WMPNetworkSvc -verbose  
VERBOSE: Performing the operation "Start-Service" on target "Windows Media Player Network (WMPNetworkSvc)".  
PS C:\users>
```

The following is a practical application of these cmdlets. The Oracle database has a number of services that must be running for the database to operate. There are times for maintenance purposes that the database must be brought down. This

requires that the services be stopped in a prescribed sequence to avoid difficulties. The script below brings down these services:

```
stop-service -name OracleDBConsoleProd  
stop-service -name OracleOraDb11g_home1TNSListener  
stop-service -name OracleServiceProd
```

Bring up these services requires reversing the order in which they were brought down. It also requires an additional step since after the Oracle database server (OracleServiceProd in our example) is started, the administrator has to log in and open the database. We add the step to start the SQLPlus process and wait until the administrator closes SQLPlus before we start the other services.

```
start-service -name OracleServiceProd  
$p = [System.Diagnostics.Process]::Start('cmd', '/c sqlplus / as sysdba')  
wait-process -id $p.id  
start-service -name OracleOraDb11g_home1TNSListener  
start-service -name OracleDBConsoleProd
```

We also have the ability to change service properties using set-service. This cmdlet is useful in disabling a service.

```
set-service -name wsearch -startuptype disabled
```

Conditional Logic

Friday, November 13, 2015
10:51 AM

A condition is a comparison expression that returns a *Boolean result*. A Boolean result is either true or false. The *conditional expression* also referred to as a *Boolean expression* is an expression that returns a Boolean result. The syntax for a conditional expression usually consists of one object compared against another using a comparison operator. The following comparison operators are all **case-insensitive** by default.

Operator	Description
-eq	Equal to
-lt	Less than
-gt	Greater than
-ge	Greater than or Equal to
-le	Less than or equal to
-ne	Not equal to
-like	Comparison to pattern with wildcards
-notlike	Opposite of -like
-match	Regular expression comparison
-notmatch	Opposite of -match
-contains	Returns true if collection contains an object
-notcontains	Opposite of -contains
-in	Identical to -contains but the operands are reversed
-notin	Opposite of -in

Prefixing the operator by an "c" make it case-insensitive when comparing against strings

The logical operators that may be used to join multiple conditional expressions to create a complex conditional expression are: -and, -or, -xor

AND	TRUE	FALSE	OR	TRUE	FALSE	XOR	TRUE
TRUE	True	False	TRUE	True	True	TRUE	False
FALSE	False	False	FALSE	True	False	FALSE	True

We may use the magnitude operators to test for equality or value that falls with certain limits.

In this example, we test for strings and numbers .

```
$s = 'abc'  
$s -eq 'ABC'      # the result is true because the default is a case-insensitive  
$s -ceq 'ABC'      # the result is false
```

```
PS C:\users>  
PS C:\users> $s = 'abc'  
PS C:\users> $s -eq 'ABC'      # the result is true because the default is a case-insensitive  
True  
PS C:\users> $s -ceq 'ABC'      # the result is false  
False  
PS C:\users> $s -lt 'ABC'      # false because of case insensitive comparison  
False  
PS C:\users> $s -clt 'ABC'      # true because 'abc' is less than 'ABC' due to ASCII collation  
True
```

We may create complex conditions to test if a value falls within a range.

```
$n = 10  
$n -gt 15  
$n -lt 20  
$n -gt 5 -and $n -lt 15  
$n -ne 6  
-not ($n -gt 5 -and $n -lt 15)
```

```
PS C:\Users>  
PS C:\Users> $n = 10  
PS C:\Users>  
PS C:\Users> $n -gt 15  
False  
PS C:\Users> $n -lt 20  
True  
PS C:\Users> $n -gt 5 -and $n -lt 15  
True  
PS C:\Users> $n -ne 6  
True  
PS C:\Users> -not ($n -gt 5 -and $n -lt 15)  
False
```

We may create complex conditional expressions that mix and match operators as below. In that example, we find all files whose size is greater than or equal to 5MB and whose name begins with "m".

```
get-childitem c:\windows | where { $_.length -ge 5MB -and $_.name -like 'm*'}
```

```
PS C:\Users>  
PS C:\Users> get-childitem c:\windows | where { $_.length -ge 5MB -and $_.name -like
```

```

Directory: C:\windows

Mode                LastWriteTime         Length Name
----              -d-----          757422262 MEMORY.DMP

```

We may also use regular expressions with `-match` and `-nomatch`. The above pipeline may be rewritten to use the `match` operator as below. The caret "`^`" is a beginning of text anchor. This means that "`m`" or "`M`" must be the first character in the matching string, the file name in the example below. The `".*"` is the equivalent of the wildcard `"*"` meaning any character multiple times. Finally the `"$"` is an end-of-text anchor. This means the "dmp" must be the last characters in the matching string.

```
get-childitem c:\windows | where { $_.length -ge 5MB -and $_.name -match '^m.*dmp' }
```

```

PS C:\Users>
PS C:\Users> get-childitem c:\windows | where { $_.length -ge 5MB -and $_.name -match '^m.*dmp' }

Directory: C:\windows

Mode                LastWriteTime         Length Name
----              -d-----          757422262 MEMORY.DMP

```

Searching collections:

```
$s = 'The quick brown fox jumps over the lazy dog'
$sa = $s.split()  #splits the sentence into an array of individual words
$sa
$sa -contains 'fox'
```

```

PS C:\Users>
PS C:\Users> $s = 'The quick brown fox jumps over the lazy dog'
PS C:\Users> $sa = $s.split()
PS C:\Users> $sa.Length
9
PS C:\Users> $sa[0]
The
PS C:\Users> $sa[8]
dog
PS C:\Users> $sa -contains 'fox'
True
PS C:\Users> $sa -contains 'cat'
False
PS C:\Users> $sa -notcontains 'fox'
False

```

```
PS C:\Users> $sa -notcontains 'cat'  
True  
PS C:\Users>
```

List specific processes:

```
get-process | where { $_.processname -contains 'svchost' }
```

```
PS C:\Users>  
PS C:\Users> get-process | where { $_.processname -contains 'svchost'  


| Handles | NPM(K) | PM(K)  | WS(K)  | VM(M) | CPU(s) | Id   | ProcessName |
|---------|--------|--------|--------|-------|--------|------|-------------|
| 521     | 28     | 198812 | 181856 | 313   |        | 120  | svchost     |
| 671     | 89     | 24012  | 13388  | 124   |        | 340  | svchost     |
| 1894    | 565    | 328640 | 183060 | 907   |        | 460  | svchost     |
| 401     | 14     | 5084   | 4540   | 44    |        | 692  | svchost     |
| 496     | 17     | 6848   | 6108   | 56    |        | 768  | svchost     |
| 548     | 26     | 19504  | 12448  | 84    |        | 992  | svchost     |
| 150     | 9      | 3740   | 2964   | 33    |        | 1048 | svchost     |
| 776     | 60     | 37608  | 19920  | 462   |        | 1356 | svchost     |
| 379     | 36     | 19208  | 10464  | 89    |        | 1492 | svchost     |
| 337     | 26     | 7552   | 5672   | 59    |        | 1600 | svchost     |
| 100     | 10     | 2672   | 884    | 31    |        | 1764 | svchost     |
| 119     | 9      | 2308   | 6484   | 37    |        | 6532 | svchost     |


```

Using wildcards:

The * wildcard will match zero or more characters

The ? wildcard will match a single character

[m-n] Match a range of characters starting from character *m* to character *n*, so [r-t]ake matches rake, sake, and take.

[abc] Match any of the characters a, b, or c; so [mt]ake matches make or take.

```
get-childitem | ? {$_ . name -like '* .txt'}
```

Using regular expressions:

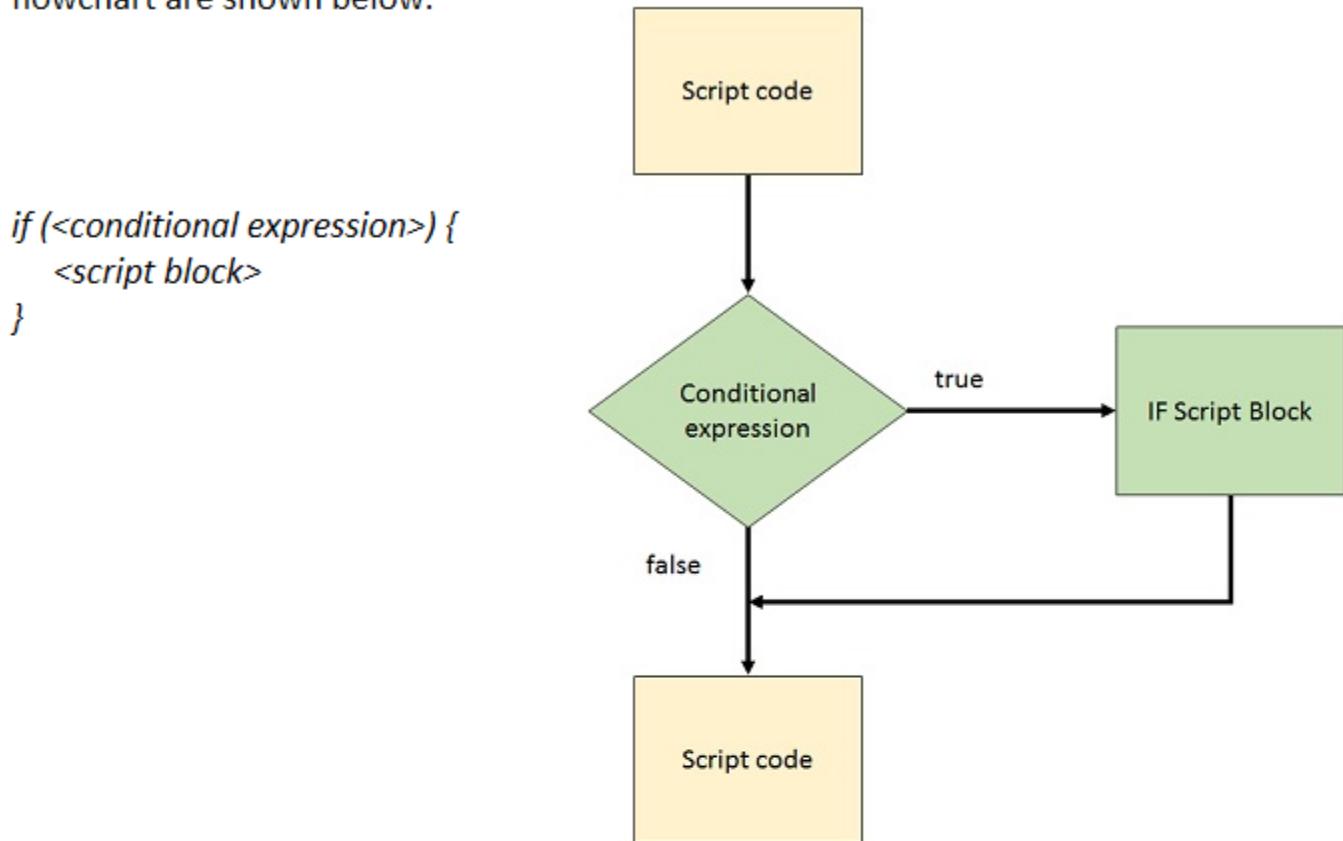
```
-----
```

```
$s = 'The quick brown fox jumps over the lazy dog'  
$s -match 'fox'          # match fox anywhere in the string  
$s -match 'quick.*fox'    # match quick, any number of characters, fox  
$s -match 'quick.*f[o|i]x' # match quick, any number of characters, fox or fix
```

IF Statement

Friday, November 13, 2015
10:54 AM

The IF statement, in its simplest form, selects or skips a path through the script or script block. In this form, the IF statement functions similar to a where-object cmdlet. The difference is that it is used in the body of the script or within a script block instead of a pipeline. The syntax and logic flowchart are shown below.



If the *<conditional expression>* evaluates to true then the script block executes otherwise it continues with the next statement. What we learned about conditional expressions with where-object apply here and other forms of IF where conditional expressions are used.

First, let's look at a few simple examples.

```
$color = 'red'
if ( $color -eq 'red' ) { write-host "The color is red" -ForegroundColor red }
if ( $color -eq 'blue' ) { write-host "The color is blue" -ForegroundColor blue }
if ( $color -ne 'blue' ) { write-host "The color is not blue" -ForegroundColor red }

$num = 17
if ( $num -gt 10 ) { write-host "the number is greater than 10" -ForegroundColor red }
if ( $num -eq 10 ) { write-host "the number is equal to 10" -ForegroundColor red }
if ( $num -eq 17 ) { write-host "the number is equal to 17" -ForegroundColor red }
if ( $num -gt 10 -and $num -lt 20 ) {
    write-host "the number is between 10 and 20 " -ForegroundColor red
}
```

```

PS C:\Users> if ( $color -eq 'red' ) { write-host "The color is red" -ForegroundColor red }
The color is red
PS C:\Users> if ( $color -eq 'blue' ) { write-host "the color is blue" -ForegroundColor blue }
PS C:\Users> if ( $color -ne 'blue' ) { write-host "The color is not blue" -ForegroundColor red }
The color is not blue
PS C:\Users> $num = 17
PS C:\Users> if ( $num -gt 10 ) { write-host "the number is greater than 10" -ForegroundColor red }
the number is greater than 10
PS C:\Users> if ( $num -eq 10 ) { write-host "the number is equal to 10" -ForegroundColor red }
PS C:\Users> if ( $num -eq 17 ) { write-host "the number is equal to 17" -ForegroundColor red }
the number is equal to 17
PS C:\Users> if ( $num -gt 10 -and $num -lt 20 ) { write-host "the number is between 10 and 20" -ForegroundColor red }
the number is between 10 and 20

```

This example is taken from the where-object cmdlet section. We implement the equivalent pipeline using an IF statement. Each fileinfo and directoryinfo object is individually piped into the foreach-object cmdlet. In this script block for that cmdlet, we implement an IF statement. The condition expression for the IF tests the property `$_._Extension` for equality to ".exe". If the expression evaluates to true, we execute the body of the IF statement where the statement `$_` passes the current object down the pipe. The foreach-object cmdlet unlike where-object does not by default pass the object down the pipe. We need the syntax shown to make that happen. When the `$_._Extension -eq '.exe'` evaluates to \$false, the object is dropped and we iterate back to the foreach-object to process the next object.

The foreach-object combined with the IF statement produces the same results as the where-object cmdlet. This example is for illustration purposes since in this case, using the where-object cmdlet is more efficient.

```

# Check that the Extension property of the file system object is equal to .exe
get-childitem c:\windows | where-object { $_._Extension -eq '.exe'}

# This accomplishes the same as the above pipeline
get-childitem c:\windows |
foreach-object {
    if ( $_._Extension -eq '.exe' ) {
        $_          # pass the object down the pipeline
    } # if the above statement is not executed the object is dropped
}

```

```

PS C:\Users>
PS C:\Users> get-childitem c:\windows | where-object { $_._Extension -eq '.exe'}

```

Directory: C:\windows

Mode	LastWriteTime	Length	Name
-a---	7/10/2015 6:59 AM	61952	bfsvc.exe
-a---	8/11/2015 6:04 AM	4532304	explorer.exe
-a---	7/10/2015 7:00 AM	994816	taskhostw.exe

```

-a---]    7/10/2015  7:00 AM      994816 HelpPane.exe
-a---]    7/10/2015  7:00 AM      18432 hh.exe
-a---]    8/15/2015  2:25 AM     215040 notepad.exe
-a---]    7/10/2015  6:59 AM     156160 regedit.exe
-a---]    7/10/2015  7:00 AM     128000 splwow64.exe
-a---]    7/10/2015  7:00 AM      10240 winhlp32.exe
-a---]    7/10/2015  7:00 AM      11264 write.exe

PS C:\Users> get-childitem c:\windows | 
>>>   foreach-object {
>>>     if ( $_.Extension -eq '.exe' ) {
>>>       $_ # pass the object down the pipeline
>>>     } # if the above statement is not executed the object is dropped
>>>   }

Directory: C:\windows

Mode                LastWriteTime         Length Name
----                -----          ----
-a---]    7/10/2015  6:59 AM      61952 bfcsvc.exe
-a---]    8/11/2015  6:04 AM     4532304 explorer.exe
-a---]    7/10/2015  7:00 AM      994816 HelpPane.exe
-a---]    7/10/2015  7:00 AM      18432 hh.exe
-a---]    8/15/2015  2:25 AM     215040 notepad.exe
-a---]    7/10/2015  6:59 AM     156160 regedit.exe
-a---]    7/10/2015  7:00 AM     128000 splwow64.exe
-a---]    7/10/2015  7:00 AM      10240 winhlp32.exe
-a---]    7/10/2015  7:00 AM      11264 write.exe

```

This first pipeline is also taken from the where-object section. We implement the equivalent pipeline using the IF statement in the second pipeline. The third pipeline reduces the syntax of IF statement by recognizing that the ISReadyOnly is a boolean property which has a value of either \$true or \$false. The example below demonstrates this concept.

```

PS C:\Users>
PS C:\Users> write-host "$($false -eq $false)" -ForegroundColor red
True
PS C:\Users> write-host "$($false -eq $true)" -ForegroundColor red
False
PS C:\Users> [boolean] $IsReadOnly = $false
PS C:\Users> write-host "$($IsReadOnly -eq $false)" -ForegroundColor red
True
PS C:\Users> write-host "$($IsReadOnly -eq $true)" -ForegroundColor red
False
PS C:\Users>

```

Since a conditional expression results in a \$true or \$false value, the comparison `$_._ReadOnly -eq $true` is unnecessary since the only possible executions of the conditional expression would be equivalent of `$true -eq $true`, which results in \$true, or `$false -eq $true`, which results in \$false. Therefore we may just drop the logical comparison and just test the `IsReadOnly` property.

```

# find the font files that have the readonly bit set
get-childitem c:\windows\fonts | where-object { $_.IsReadOnly -eq $true}

# accomplishes the same as the above

```

```

get-childitem c:\windows\fonts |
    foreach-object {
        if ( $_.Isreadonly -eq $true ) {
            $_
        }
    }

# since Isreadonly is a boolean it has a value of $true or false
get-childitem c:\windows\fonts |
    foreach-object {
        if ( $_.Isreadonly ) {
            $_
        }
    }

```

```

PS C:\Users>
PS C:\Users> # find the font files that have the readonly bit set
PS C:\Users> get-childitem c:\windows\fonts | where-object { $_.Isreadonly -eq $true}

    Directory: C:\windows\Fonts

Mode                LastWriteTime      Length Name
----              -----          ----  --
-a---       11/30/2014 12:13 AM        7276 TSPECIAL1.TTF

PS C:\Users>
PS C:\Users> # accomplishes the same as the above
PS C:\Users> get-childitem c:\windows\fonts |
>>>     foreach-object {
>>>         if ( $_.Isreadonly -eq $true ) {
>>>             $_
>>>         }
>>>     }

    Directory: C:\windows\Fonts

Mode                LastWriteTime      Length Name
----              -----          ----  --
-a---       11/30/2014 12:13 AM        7276 TSPECIAL1.TTF

PS C:\Users>
PS C:\Users> # since Isreadonly is a boolean it has a value of $true or false
PS C:\Users> get-childitem c:\windows\fonts |
>>>     foreach-object {
>>>         if ( $_.Isreadonly ) {
>>>             $_
>>>         }
>>>     }

    Directory: C:\windows\Fonts

Mode                LastWriteTime      Length Name
----              -----          ----  --
-a---       11/30/2014 12:13 AM        7276 TSPECIAL1.TTF

```

In the above examples, the use of the IF statement is not as efficient as using the where-object cmdlet. In this example, that is not the case. What is illustrated is the most efficient way to accomplish the goals of the next example based upon what we know.

```
-----+-----+-----+-----+-----+-----+-----+-----+
```

This example uses the .csv file below. The goal of this script is to count the number packets defined for port 80 and port 8080. This file is saved a firewall_log.csv.

Date	Time	Syslog_ID	Source_IP	Source_Port	Dest_IP	Dest_Port
Sep 16 2015	14:54:31	106023	188.26.62.236	30870	207.75.134.154	443
Sep 17 2015	14:54:31	106023	193.110.18.33	62629	207.75.134.10	80
Sep 16 2015	14:54:30	106023	187.13.238.91	22350	207.75.134.125	995
Sep 17 2015	14:54:29	106023	46.249.171.169	27376	207.75.134.154	8080
Sep 16 2015	14:54:29	106023	188.134.40.93	49611	207.75.134.80	443
Sep 17 2015	14:54:29	106023	187.13.238.91	13757	207.75.134.154	80
Sep 16 2015	14:54:29	106023	80.99.73.186	16008	207.75.134.100	995
Sep 17 2015	14:54:29	106023	187.13.238.91	48884	207.75.134.154	8080
Sep 16 2015	14:54:31	106023	188.26.62.236	30870	207.75.134.154	443
Sep 17 2015	14:54:31	106023	193.110.18.33	62629	207.75.134.10	80
Sep 16 2015	14:54:30	106023	187.13.238.91	22350	207.75.134.125	995
Sep 17 2015	14:54:29	106023	46.249.171.169	27376	207.75.134.154	8080
Sep 16 2015	14:54:29	106023	188.134.40.93	49611	207.75.134.80	443
Sep 17 2015	14:54:29	106023	187.13.238.91	13757	207.75.134.154	80
Sep 16 2015	14:54:29	106023	80.99.73.186	16008	207.75.134.100	995
Sep 17 2015	14:54:29	106023	187.13.238.91	48884	207.75.134.154	8080

In this example, we first initialize the variables we are using to make the script repeatable. Each of the firewall log is piped into the foreach-object script block. In the script block of the IF, we test the dest_port property. If the value of that property is 80, we increment the port 80 counter. The next IF statement tests for equality to 8080, if that conditional expression evaluates to \$true, we increment the port 8080 counter. In each of the script blocks of the IF statements, we pass the current object down the pipeline by using the statement \$. This causes the object to exit the pipeline and be subsequently displayed in the console by PowerShell.

```
$port80 = 0
$port8080 = 0
import-csv .\firewall_log.csv |
    foreach-object {
        if ( $_.dest_port -eq 80 ) { $port80++ }
        if ( $_.dest_port -eq 8080 ) { $port8080++ }
    }
write-host
write-host "`t $port80 packets to port 80 `n`t $port8080 packets to port 8080"
write-host
```

```
2 $port80 = 0
3 import-csv .\firewall_log.csv |
4 foreach-object {
5     if ( $_.dest_port -eq 80 ) { $port80++ }
6     if ( $_.dest_port -eq 8080 ) { $port8080++ }
7 }
8 write-host
9 write-host "`t $port80 packets to port 80 `n`t $port8080 packets to port 8080"
10 write-host
```

```
<
PS C:\Users>
PS C:\Users> C:\Users\count_fw.ps1

    4 packets to port 80
    4 packets to port 8080
```

In this next example, we display and count the number of disk errors in the system log. We first initialize the variables, in this case only \$nerrs. We use get-eventlog to retrieve all events then pipe those events individually into where-object where only the objects (entries) with a Source property value of 'Disk' are selected and piped into foreach-object. In the body of the script block, we use an IF statement to select only those objects. If the source is the Disk process, we increment the \$nerrs counter to count the object and then pass the object on using the syntax \$. At the end of the script, we output the total number of errors found.

```
$nerrs = 0
get-eventlog -log system | where-object { $_.Source -eq 'Disk' } |
foreach-object {
    if ( $_.EntryType -eq 'Error' ) {
        $nerrs = $nerrs + 1
        $_      # pass object down the pipeline
    }
}
write-host "`n`n Disk errors the system log:"
write-host "`t`t $nerrs`terror entries found"
```

```
1 $nerrs = 0
2 get-eventlog -log system | where-object { $_.Source -eq 'Disk' } |
3 foreach-object {
4     if ( $_.EntryType -eq 'Error' ) {
5         $nerrs = $nerrs + 1
6         $_
7     }
8 }
9 write-host "`n`n Disk errors the system log:"
10 write-host "`t`t $nerrs`terror entries found"
```

```
<
PS C:\Users>
PS C:\Users> C:\Users\count_events1.ps1
```

Index	Time	EntryType	Source	InstanceID	Message
64013	Sep 18 22:43	Error	disk	3221487623	The device, \Device\
63865	Sep 18 22:23	Error	disk	3221487623	The device, \Device\
63864	Sep 18 22:23	Error	disk	3221487623	The device, \Device\
63849	Sep 18 22:17	Error	disk	3221487623	The device, \Device\
63845	Sep 18 22:11	Error	disk	3221487623	The device, \Device\
63743	Sep 18 04:52	Error	disk	3221487623	The device, \Device\

Disk errors the system log:
6 error entries found

What if we would like to run this script with a different log or different event source. We would need to change the get-eventlog cmdlet replacing the literal system with the name of the new log. We would also need to replace the literal 'Disk' in the script block of the where-object cmdlet. Finally, we need to change the write-host statements to reflect these changes. Every change to the script could inadvertently introduce a bug. So, we want to make our script more flexible and reduce the number of changes by using variables instead of literals.

What is shown below is a common technique. We create the variable \$log and assign it the value 'system'. We also create the variable \$source and assign it the value 'Disk'. We next replace every occurrence of 'system' and 'Disk' with the appropriate variable. It should now be very obvious that we need to run this script against another log, event source or both, the only change required is to the variables, the remainder of the script would remain untouched.

```
$nerrs = 0
$log = 'system'
$source = 'Disk'
get-eventlog -log $log | where-object { $_.Source -eq $source } |
    foreach-object {
        if ( $_.EntryType -eq 'Error' ) {
            $nerrs = $nerrs + 1
            $_
        }
    }
write-host "`n`n $source errors the $log log:"
write-host "`t`t $nerrs`terror entries found"
```

```
1 $nerrs = 0
2 $log = 'system'
3 $source = 'Disk'
4 get-eventlog -log $log | where-object { $_.Source -eq $source } |
5   foreach-object {
6     if ( $_.EntryType -eq 'Error' ) {
7       $nerrs = $nerrs + 1
8       $_
9     }
}
```

```
10  }
11  write-host "`n`n $source errors the $log log:"
12  write-host "`t`t $nerrs`terror entries found"
```

```
<
PS C:\Users>
PS C:\Users> C:\Users\count_events1.ps1
```

Index	Time	EntryType	Source
-----	-----	-----	-----
64013	Sep 18 22:43	Error	disk
63865	Sep 18 22:23	Error	disk
63864	Sep 18 22:23	Error	disk
63849	Sep 18 22:17	Error	disk
63845	Sep 18 22:11	Error	disk
63743	Sep 18 04:52	Error	disk

InstanceID	Message
-----	-----
3221487623	The device, \Device\HAR

```
Disk errors the system log:
 6 error entries found
```

We also like to include warnings in our output. We modify our script to include counting warnings. We do this adding another IF statement to test for an event entry type of 'warning'. We also add the write-host statements to display the warning count.

```
$nerrs = 0
$log = 'system'
$source = 'Disk'
get-eventlog -log $log | where-object { $_.Source -eq $source } |
  foreach-object {
    if ( $_.EntryType -eq 'Error' ) {
      $nerrs = $nerrs + 1
      $_
    }
    if ( $_.EntryType -eq 'Warning' ) {
      $nwarns = $nwarns + 1
      $_
    }
  }
write-host "`n`n $source errors the $log log:"
write-host "`t`t $nerrs`terror entries found"
```

Since this script is becoming more complex, the span of the script block for the foreach-object is identified by the purple arrow and similarly the span of the script blocks for the two IF statements are identified by green and blue arrows.

```
1  $nerrs = 0
2  $log = 'system'
3  $source = 'Disk'
```

```

4  get-eventlog -log $log | where-object { $_.Source -eq $source } |
5  foreach-object {
6      if ( $_.EntryType -eq 'Error' ) {
7          $nerrs = $nerrs + 1
8          $_
9      }
10     if ( $_.EntryType -eq 'Warning' ) {
11         $nwarns = $nwarns + 1
12         $_
13     }
14 }
15 write-host "`n`n $source errors and warnings for the $log log:"
16 write-host "`t`t $nerrs`terror entrys found"
17 write-host "`t`t $nwarns`twarning entrys found"

```

```

PS C:\Users>
PS C:\Users> C:\Users\count_events1.ps1

```

Index	Time	EntryType	Source	InstanceID	Message
64168	Sep 19 07:39	Warning	disk	2147745843	An error was detected on de
64167	Sep 19 07:39	Warning	disk	2147745843	An error was detected on de
64165	Sep 19 07:38	Warning	disk	2147745945	The IO operation at logical
64163	Sep 19 07:36	Warning	disk	2147745945	The IO operation at logical
64161	Sep 19 07:34	Warning	disk	2147745945	The IO operation at logical
64160	Sep 19 07:32	Warning	disk	2147745945	The IO operation at logical
64013	Sep 18 22:43	Error	disk	3221487623	The device, \Device\Harddi
63865	Sep 18 22:23	Error	disk	3221487623	The device, \Device\Harddi
63864	Sep 18 22:23	Error	disk	3221487623	The device, \Device\Harddi
63852	Sep 18 22:19	Warning	disk	2147745945	The IO operation at logical
63850	Sep 18 22:17	Warning	disk	2147745945	The IO operation at logical
63849	Sep 18 22:17	Error	disk	3221487623	The device, \Device\Harddi
63848	Sep 18 22:15	Warning	disk	2147745945	The IO operation at logical
63847	Sep 18 22:13	Warning	disk	2147745945	The IO operation at logical
63846	Sep 18 22:12	Warning	disk	2147745945	The IO operation at logical
63845	Sep 18 22:11	Error	disk	3221487623	The device, \Device\Harddi
63743	Sep 18 04:52	Error	disk	3221487623	The device, \Device\Harddi

```

Disk errors and warnings for the system log:
  6  error entrys found
  22 warning entrys found

```

Inspecting this script, we notice the number of the times that a log entry is process. The get-eventlog cmdlet process every entry in the system log. The where-object also process every entry likewise for the foreach-object cmdlet. Assuming the system eventlog contains 2000 entries, a rather modest number, the total number of iterations once this script is complete is 2000 (get-eventlog) + 2000 (where-object) + 2000 (foreach-object) \rightarrow 6000 entries. We would like to make this script more efficient. The way to do this is to eliminate the where-object cmdlet.

This variation of the script implements a structure commonly called the *nested IF*.

We first initialize the variable used in the script. In the pipeline we retrieve all of the entries i

log referenced by \$log, which has a value of system. Each entry is piped individually to the foreach-object cmdlet. In the script block for this cmdlet, the first IF statement, delimited by the red arrow, tests the Source property to determine if it is a desired source as specified by the variable \$source. If the conditional expression \$_.Source -eq \$source evaluates to false, we drop the object and return back to the beginning of the pipeline to process the next object.

If the conditional expression of the outer (red) IF evaluates to true, we execute the script block of the outer (red) IF. In that script block, we implement same two IF statements (green and blue arrows) to test the entry type of the event. The first (green) IF statement in the script block tests for an entry type of 'Error'. If this conditional expression evaluates to true, we increments the error count \$nerrs and passes the object down the pipe with the statement \$_. In a similar manner, the second (blue) IF statement tests for warnings. If the entry type property of the current object is not processed by either IF statement, the object is dropped and control is returned back to the beginning of the pipeline where the next object is processed. Recall that foreach-object does not by default pass the object to the next step in the pipeline.

```
$nerrs = 0
$nwarns = 0
$log = 'system'
$source = 'Disk'
get-eventlog -log $log |
foreach-object {
    if ( $_.Source -eq $source ) {
        if ( $_.EntryType -eq 'Error' ) {
            $nerrs = $nerrs + 1
            $_
        }
        if ( $_.EntryType -eq 'Warning' ) {
            $nwarns = $nwarns + 1
            $_
        }
    }
}
write-host "`n`n $source errors and warnings for the $log log:"
write-host "`t`t $nerrs`terror entries found"
write-host "`t`t $nwarns`twarning entries found"
```

```
1 $nerrs = 0
2 $nwarns = 0
3 $log = 'system'
4 $source = 'Disk'
5 get-eventlog -log $log |
6 →foreach-object {
7   if ( $_.Source -eq $source ) {
8     if ( $_.EntryType -eq 'Error' ) {
9       $nerrs = $nerrs + 1
10      $_
```

```

11 }
12 if ( $_.EntryType -eq 'Warning' ) {
13     $nwarns = $nwarns + 1
14     $_
15 }
16 }
17 }

18 write-host "`n`n $source errors and warnings for the $log log:"
19 write-host "'`t`t $nerrs`terror entrys found"
20 write-host "'`t`t $nwarns`twarning entrys found"

```

<

```

PS C:\Users>
PS C:\Users> C:\Users\count_events.ps1
```

Index	Time	EntryType	Source	InstanceID	Message
64168	Sep 19 07:39	Warning	disk	2147745843	An error was detected on device \
64167	Sep 19 07:39	Warning	disk	2147745843	An error was detected on device \
64165	Sep 19 07:38	Warning	disk	2147745945	The IO operation at logical block
64163	Sep 19 07:36	Warning	disk	2147745945	The IO operation at logical block
64161	Sep 19 07:34	Warning	disk	2147745945	The IO operation at logical block
64160	Sep 19 07:32	Warning	disk	2147745945	The IO operation at logical block
64013	Sep 18 22:43	Error	disk	3221487623	The device, \Device\Harddisk4\DR4
63865	Sep 18 22:23	Error	disk	3221487623	The device, \Device\Harddisk4\DR8
63864	Sep 18 22:23	Error	disk	3221487623	The device, \Device\Harddisk4\DR8
63852	Sep 18 22:19	Warning	disk	2147745945	The IO operation at logical block
63850	Sep 18 22:17	Warning	disk	2147745945	The IO operation at logical block
63849	Sep 18 22:17	Error	disk	3221487623	The device, \Device\Harddisk4\DR4
63848	Sep 18 22:15	Warning	disk	2147745945	The IO operation at logical block
63847	Sep 18 22:13	Warning	disk	2147745945	The IO operation at logical block
63846	Sep 18 22:12	Warning	disk	2147745945	The IO operation at logical block
63845	Sep 18 22:11	Error	disk	3221487623	The device, \Device\Harddisk4\DR4
63743	Sep 18 04:52	Error	disk	3221487623	The device, \Device\Harddisk4\DR4

```

Disk errors and warnings for the system log:
    6  error entrys found
    11 warning entrys found

```

In the above script, we used two variables \$log and \$source to, respectively, hold the value of log name and the name of the source that produce the entry. Those variable are referenced in pipeline and in the write-host statements. If we desired to reference a different log or a different source or both, we simply just replace the literal string in the assignment statement in the beginning of the script .

Because of this design, we may yet again create a more flexible variation by passing the log name and the source as parameters from the command line. So now, when we need to be run against a different log or event source, we simply pass them as parameters.

The changes to accomplish this as shown below are trivial. We simply add a param statement for the two parameters that were previously variables in the script.

```

1 param ( [string] $log, [string] $source )
2 $nerrs = 0

```

The param statement replaces the assignment statements.

```

3 $nwarns = 0
4 # $log = 'system'
5 # $source = 'Disk'
6 get-eventlog -log $log |
7 foreach-object {
8 if ( $_.Source -eq $source ) {
9   if ( $_.EntryType -eq 'Error' ) {
10     $nerrs = $nerrs + 1
11     $_
12   }
13   if ( $_.EntryType -eq 'Warning' ) {
14     $nwarns = $nwarns + 1
15     $_
16   }
17 }
18 }
19 write-host "`n`n $source errors and warnings for the $log log:"
20 write-host "`t`t $nerrs`terror entrys found"

```

These assignment statements are commented out. They may be removed without any impact.

```

PS C:\Users>
PS C:\Users> .\count_events.ps1 'system' 'disk'

```

Index	Time	EntryType	Source	InstanceID	Message
64168	Sep 19 07:39	Warning	disk	2147745843	An error was detected on device \Device\Harddisk4\DR4, logical block address 0x0000000000000000. The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
64167	Sep 19 07:39	Warning	disk	2147745843	An error was detected on device \Device\Harddisk4\DR4, logical block address 0x0000000000000000. The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
64165	Sep 19 07:38	Warning	disk	2147745945	The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
64163	Sep 19 07:36	Warning	disk	2147745945	The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
64161	Sep 19 07:34	Warning	disk	2147745945	The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
64160	Sep 19 07:32	Warning	disk	2147745945	The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
64013	Sep 18 22:43	Error	disk	3221487623	The device, \Device\Harddisk4\DR4, logical block address 0x0000000000000000, failed to respond to a read command.
63865	Sep 18 22:23	Error	disk	3221487623	The device, \Device\Harddisk4\DR8, logical block address 0x0000000000000000, failed to respond to a read command.
63864	Sep 18 22:23	Error	disk	3221487623	The device, \Device\Harddisk4\DR8, logical block address 0x0000000000000000, failed to respond to a read command.
63852	Sep 18 22:19	Warning	disk	2147745945	The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
63850	Sep 18 22:17	Warning	disk	2147745945	The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
63849	Sep 18 22:17	Error	disk	3221487623	The device, \Device\Harddisk4\DR4, logical block address 0x0000000000000000, failed to respond to a read command.
63848	Sep 18 22:15	Warning	disk	2147745945	The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
63847	Sep 18 22:13	Warning	disk	2147745945	The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
63846	Sep 18 22:12	Warning	disk	2147745945	The IO operation at logical block address 0x0000000000000000 failed because the device was not ready.
63845	Sep 18 22:11	Error	disk	3221487623	The device, \Device\Harddisk4\DR4, logical block address 0x0000000000000000, failed to respond to a read command.
63743	Sep 18 04:52	Error	disk	3221487623	The device, \Device\Harddisk4\DR4, logical block address 0x0000000000000000, failed to respond to a read command.

```

disk errors and warnings for the system log:
  6  error entrys found
  11 warning entrys found

```

```

PS C:\Users>
PS C:\Users> .\count_events.ps1 'application' '.Net Runtime'

```

Index	Time	EntryType	Source	InstanceID	Message
3396	Sep 09 21:18	Error	.NET Runtime	1023	Application: mscore.dll, Version: 5.0.2900.17402, Time: 2010-08-31 18:54:00
3395	Sep 09 21:18	Error	.NET Runtime	1023	Application: mscore.dll, Version: 5.0.2900.17402, Time: 2010-08-31 18:54:00
3394	Sep 09 21:18	Error	.NET Runtime	1023	Application: mscore.dll, Version: 5.0.2900.17402, Time: 2010-08-31 18:54:00
3393	Sep 09 21:18	Error	.NET Runtime	1023	Application: mscore.dll, Version: 5.0.2900.17402, Time: 2010-08-31 18:54:00
2487	Aug 31 18:54	Error	.NET Runtime	1023	Application: mscore.dll, Version: 5.0.2900.17402, Time: 2010-08-31 18:54:00
2486	Aug 31 18:54	Error	.NET Runtime	1023	Application: mscore.dll, Version: 5.0.2900.17402, Time: 2010-08-31 18:54:00
2485	Aug 31 18:54	Error	.NET Runtime	1023	Application: mscore.dll, Version: 5.0.2900.17402, Time: 2010-08-31 18:54:00
2484	Aug 31 18:54	Error	.NET Runtime	1023	Application: mscore.dll, Version: 5.0.2900.17402, Time: 2010-08-31 18:54:00

```
.Net Runtime errors and warnings for the application log:
```

```
  8  error entrys found
```

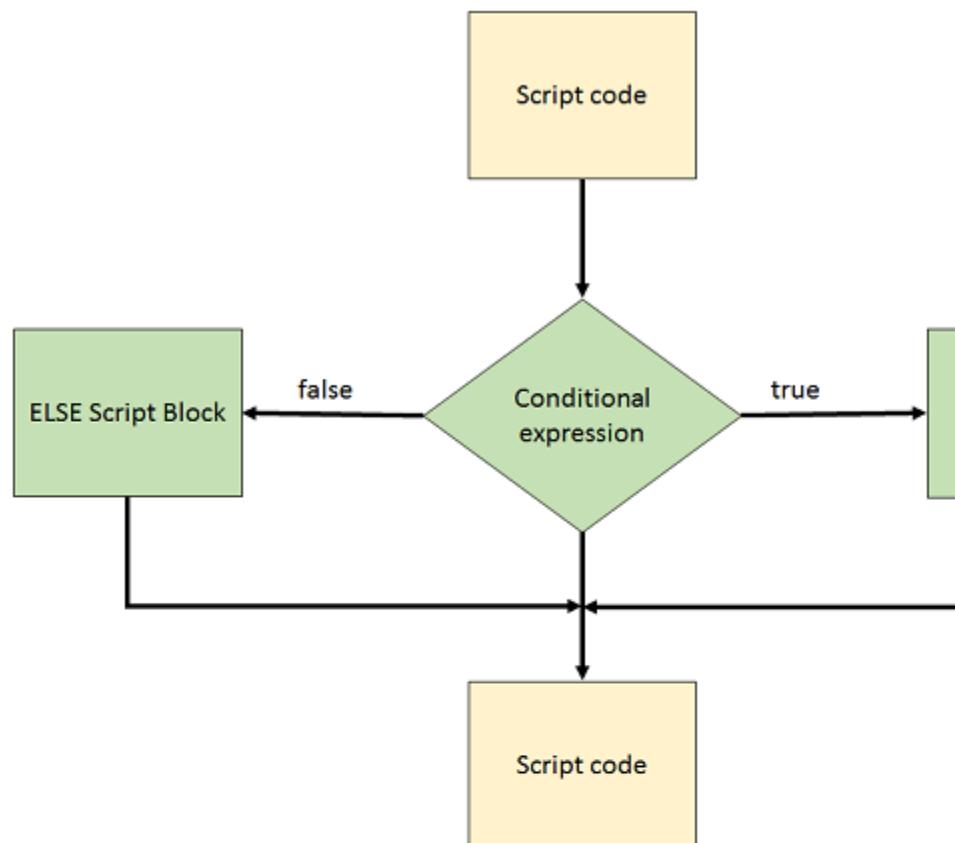
```
  0  warning entrys found
```

IF-ELSE Statement

Friday, November 13, 2015
10:55 AM

The IF-ELSE statement is a variation on the IF that provides one of two alternative paths through a script.

```
if (<conditional expression>) {  
    <IF script block>  
} else {  
    <ELSE script block>  
}
```



If the <conditional expression> evaluates to true then <IF script block > is executed. After <IF script block> completes, execution continues after the IF-ELSE statement. If <conditional expression> evaluates to false then <ELSE scriptblock> is executed. After <ELSE script block> completes, execution continues after the IF-ELSE statement.

A few simple examples demonstrate how this statement works.

```
$color = 'red'  
if ( $color -eq 'red' ) {  
    write-host "The color is red" -ForegroundColor red  
} else {  
    write-host "The color is not red" -ForegroundColor red  
}  
  
if ( $color -eq 'blue' ) {  
    write-host "The color is blue" -ForegroundColor blue  
} else {  
    write-host "The color is not blue" -ForegroundColor blue  
}  
  
if ( $color -ne 'blue' ) {  
    write-host "The color is not blue" -ForegroundColor red  
} else {  
    write-host "The color is blue" -ForegroundColor red
```

```
}

$num = 17
if ( $num -gt 10 ) {
    write-host "the number is greater than 10" -ForegroundColor red
} else {
    write-host "the number is not greater than 10" -ForegroundColor red
}

if ( $num -eq 17 ) {
    write-host "the number is equal to 17" -ForegroundColor red
} else {
    write-host "the number is not equal to 17" -ForegroundColor red
}

if ( $num -gt 10 -and $num -lt 20 ) {
    write-host "the number is between 10 and 20 " -ForegroundColor red
} else {
    write-host "the number is less than 10 or greater than 20 " -ForegroundColor red
}
```

```
PS C:\Users>
PS C:\Users> $color = 'red'
PS C:\Users> if ($color -eq 'red') {
>>>     write-host "The color is red" -ForegroundColor red
>>> } else {
>>>     write-host "The color is not red" -ForegroundColor red
>>> }
The color is red
PS C:\Users>
PS C:\Users> if ($color -eq 'blue') {
>>>     write-host "The color is blue" -ForegroundColor blue
>>> } else {
>>>     write-host "The color is not blue" -ForegroundColor blue
>>> }
The color is not blue
PS C:\Users>
PS C:\Users> if ($color -ne 'blue') {
>>>     write-host "The color is not blue" -ForegroundColor red
>>> } else {
>>>     write-host "The color is blue" -ForegroundColor red
>>> }
The color is not blue
PS C:\Users>
PS C:\Users> $num = 17
PS C:\Users> if ( $num -gt 10 ) {
>>>     write-host "the number is greater than 10" -ForegroundColor red
>>> } else {
>>>     write-host "the number is not greater than 10" -ForegroundColor red
>>> }
the number is greater than 10
PS C:\Users>
PS C:\Users> if ( $num -eq 17 ) {
>>>     write-host "the number is equal to 17" -ForegroundColor red
>>> } else {
>>>     write-host "the number is not equal to 17" -ForegroundColor red
>>> }
the number is equal to 17
PS C:\Users>
```

```
PS C:\Users> if ($num -gt 10 -and $num -lt 20) {  
>>>     write-host "the number is between 10 and 20" -ForegroundColor red  
>>> } else {  
>>>     write-host "the number is less than 10 or greater than 20" -ForegroundColor green  
>>> }  
the number is between 10 and 20
```

In this example, the script uses test-connection to ping an IP address. If the ping is successful, display a success message otherwise a failed message.

In the script below, the test-connection cmdlet returns a win32_pingstatus object with numerous properties. The property StatusCode provides the result of the ping. A value of 0 means that it was successful. A null value means that the ping was unsuccessful. The conditional expression \$alive.StatusCode tests for a successful ping. If the conditional expression results in a value of 0 (purple) IF script block executes and the success message is written to the console. If the ping is unsuccessful, meaning that the StatusCode is something other than 0, the (orange) ELSE script block executes and the failed message displayed.

```
$ipaddr = '207.75.134.64'  
$alive = test-connection -computername $ipaddr -Count 1 -ErrorAction SilentlyContinue  
if ( $alive.StatusCode -eq 0 ) {  
    write-host "`t $($alive.Address) $($alive.StatusCode) is alive"  
} else {  
    write-host "`t $ipaddr $($alive.StatusCode) is dead"  
}
```

```
1 $ipaddr = '207.75.134.64'  
2 $alive = test-connection -computername $ipaddr -Count 1 -ErrorAction SilentlyContinue  
3 if ( $alive.StatusCode -eq 0 ) {  
4     write-host "`t $($alive.Address) $($alive.StatusCode) is alive"  
5 } else {  
6     write-host "`t $ipaddr $($alive.StatusCode) is dead"  
7 }
```



```
PS C:\Users>  
PS C:\Users> .\enum1.ps1  
207.75.134.64 0 is alive
```

We alter the above script so then when a successful ping occurs, we attempt to resolve the IP address to a host name.

We alter the above script by adding to the (purple) IF script block the statement to resolve the IP address to a host name using the resolve-dnsname cmdlet. The result of that cmdlet is stored in variable \$dnsresult. The (green) IF statement checks value of \$dnsresult. If the variable is not null, then the host name resolution was successful so we may display the appropriate message with the host name.

variable \$dnsresult having a null value means that the cmdlet resolve-dnsresult did not return because there is no "A" record for the IP address, i.e., the host name is not registered with DNS.

```
$ipaddr = '207.75.134.64'  
$alive = test-connection -computername $ipaddr -Count 1 -ErrorAction SilentlyContinue  
if ( $alive.statuscode -eq 0 ) {  
  
    $dnsresult = resolve-dnsname $alive.address -ErrorAction SilentlyContinue  
    write-host "`t $($alive.address) $($alive.statuscode) is alive"  
  
    if ( $dnsresult -ne $null ) {  
        write-host "`t`t and has the name $($dnsresult.namehost)"  
    }  
  
} else {  
    write-host "`t $ipaddr $($alive.statuscode) is dead"  
}  
  
1 $ipaddr = '207.75.134.64'  
2 $alive = test-connection -computername $ipaddr -Count 1 -ErrorAction SilentlyContinue  
3 If ( $alive.statuscode -eq 0 ) {  
4  
5     $dnsresult = resolve-dnsname $alive.address -ErrorAction SilentlyContinue  
6     write-host "`t $($alive.address) $($alive.statuscode) is alive"  
7  
8     if ( $dnsresult -ne $null ) {  
9         write-host "`t`t and has the name $($dnsresult.namehost)"  
10    }  
11  
12 } else {  
13     write-host "$ipaddr $($alive.statuscode) is dead"  
14 }
```

```
PS C:\Users>  
PS C:\Users> C:\Users\test-conn.ps1  
207.75.134.64 0 is alive  
and has the name wcc3-64.wccnet.org
```

We would like to make this script more flexible; so that instead of running the script against one host we want to ping every host in a given class C subnet. To do this we need to ping (test-connection) every host starting with host 1 and ending with host 254. We will pass the first three octets of the subnet parameter to the script.

To accomplish our goal, we use the range operator ".." to generate the range 1..254. Each integer is piped to (red) foreach-object whose script block is identical to the previous script.

This function of the script is called a *ping sweep* and is commonly used in penetration testing to enumerate all hosts in a subnet.

```

param ( [string] $subnet3 )

1..254 | foreach-object {
    $ipaddr = $subnet3 + '.' + $_ # append "." and the current host number
    $alive =
        test-connection -computername $ipaddr -Count 1 -ErrorAction SilentlyContinue
    if ( $alive.statuscode -eq 0 ) {

        $dnsresult = resolve-dnsname $alive.address -ErrorAction SilentlyContinue
        write-host "`t $($alive.address) $($alive.statuscode) is alive"

        if ( $dnsresult -ne $null ) {
            write-host "`t`t and has the name $($dnsresult.namehost)"
        }
    } else {
        write-host "`t $ipaddr $($alive.statuscode) is dead"
    }
}

```

```

1 param ( [string] $subnet3 )
2
3 1..254 | foreach-object {
4     $ipaddr = $subnet3 + '.' + $_ # append "." and the current host number
5     $alive =
6         test-connection -computername $ipaddr -Count 1 -ErrorAction SilentlyContinue
7     if ( $alive.statuscode -eq 0 ) {
8
9         $dnsresult = resolve-dnsname $alive.address -ErrorAction SilentlyContinue
10        write-host "`t $($alive.address) $($alive.statuscode) is alive"
11
12         if ( $dnsresult -ne $null ) {
13             write-host "`t`t and has the name $($dnsresult.namehost)"
14         }
15     } else {
16         write-host "`t $ipaddr $($alive.statuscode) is dead"
17     }
18 }
19
20

```

```

PS C:\Users>
PS C:\Users> .\enum-subnet.ps1 '207.75.134'
207.75.134.1 0 is alive

```

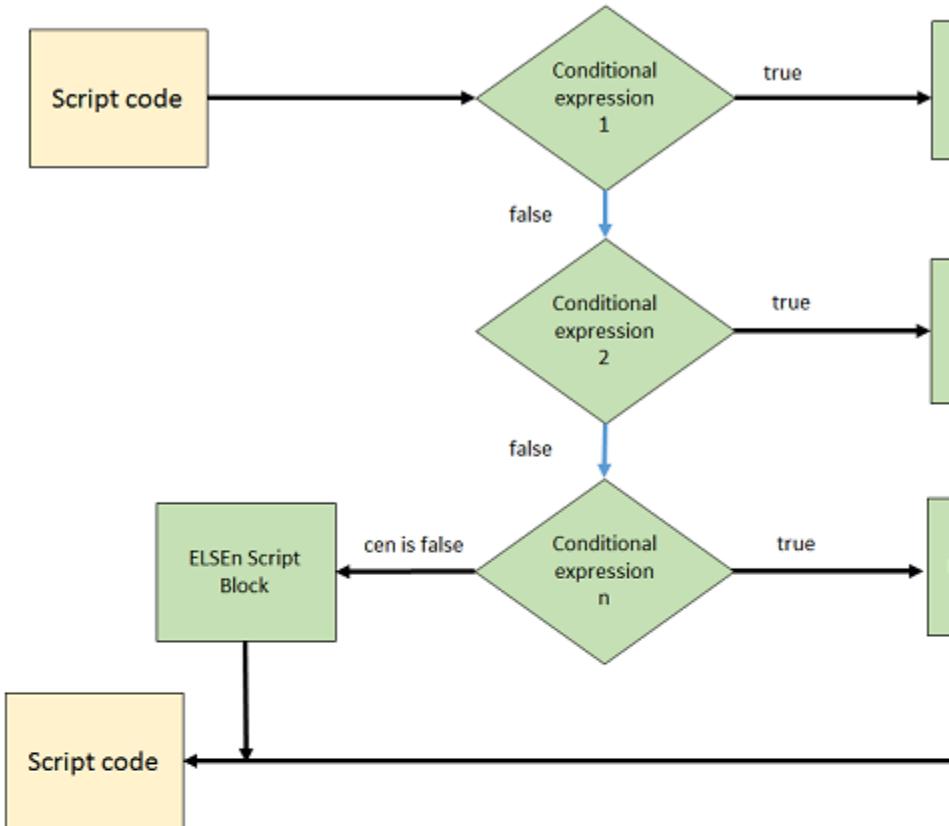
```
207.75.134.2  is dead
207.75.134.3  is dead
207.75.134.4  is dead
207.75.134.5  is dead
207.75.134.6 0 is alive
    and has the name cssgate1.wccnet.org
207.75.134.7 0 is alive
    and has the name cssgate2.wccnet.org
207.75.134.8  is dead
207.75.134.9  is dead
207.75.134.10 is dead
207.75.134.11 0 is alive
    and has the name wcc3-11.wccnet.org
207.75.134.12  is dead
207.75.134.13 0 is alive
    and has the name wcc3-13.wccnet.org
207.75.134.14  is dead
207.75.134.15 0 is alive
    and has the name wcc3-15.wccnet.org
207.75.134.16  is dead
207.75.134.17 0 is alive
    and has the name wcc3-17.wccnet.org
207.75.134.18  is dead
```

IF-ELSEIF Statement

Friday, November 13, 2015
10:55 AM

The IF-ELSEIF statement joins a series of IF-ELSE statements. This form is used when testing for conditions where each condition is mutually exclusive from any of the others. The syntax and flowchart are shown below.

```
if (<condition1>) {
    <scriptblock1>
} elseif (<condition2>) {
    <scriptblock2>
} elseif (<condition3>) {
...
} else {
    <scriptblockn>
}
```



If the <condition expression 1> evaluates to true then <IF1 script block> is executed. After <IF1 script block> completes, execution continues after the IF-ELSEIF statement. If <condition expression 1> is false <condition expression 2> is evaluated. If <condition expression 2> is true then <IF2 script block> is executed. After <IF2 script block> completes, execution continues after the IF-ELSEIF statement. This process repeats until all conditions are evaluated. If the last <conditional expression n> evaluates to true then <scriptblockn> is executed. If <conditional expression n> evaluates to false then <ELSEN script block> is executed.

This simple example is demonstrates the use of IF-ELSEIF. A string array consisting of six colors is piped into foreach-object. In the script block, the IF-ELSEIF tests the current object (string) in the pipeline. It displays an appropriate message in the appropriate color. Note the use of the *this* object symbol, write-host message and ForegroundColor parameter.

```
clear-host
write-host
'red', 'green', 'blue', 'cyan', 'yellow', 'orange' | foreach-object {
    if ( $_ -eq 'red' ) {
        write-host "The color is $_" -ForegroundColor $_
    } elseif ( $_ -eq 'green' ) {
        write-host "The color is $_" -ForegroundColor $_
    } elseif ( $_ -eq 'blue' ) {
        write-host "The color is $_" -ForegroundColor $_
    } elseif ( $_ -eq 'cyan' ) {
        write-host "The color is $_" -ForegroundColor $_
    } elseif ( $_ -eq 'yellow' ) {
        write-host "The color is $_" -ForegroundColor $_
    } elseif ( $_ -eq 'orange' ) {
        write-host "The color is $_" -ForegroundColor $_
    }
}
```

```

} elseif ( $_ -eq 'cyan' ) {
    write-host "The color is $_" -ForegroundColor $_
} elseif ( $_ -eq 'yellow' ) {
    write-host "The color is $_" -ForegroundColor $_
} else {
    write-host "The color cannot be determined" -ForegroundColor white
}
}
write-host

1  clear-host
2  write-host
3  'red', 'green', 'blue', 'cyan', 'yellow', 'orange' | foreach-object {
4  if ( $_ -eq 'red' ) {
5      write-host "The color is $_" -ForegroundColor $_
6  } elseif ( $_ -eq 'green' ) {
7      write-host "The color is $_" -ForegroundColor $_
8  } elseif ( $_ -eq 'blue' ) {
9      write-host "The color is $_" -ForegroundColor $_
10 } elseif ( $_ -eq 'cyan' ) {
11     write-host "The color is $_" -ForegroundColor $_
12 } elseif ( $_ -eq 'yellow' ) {
13     write-host "The color is $_" -ForegroundColor $_
14 } else {
15     write-host "The color cannot be determined" -ForegroundColor white
16 }
17 }
18 write-host

```

```

The color is red
The color is green
The color is blue
The color is cyan
The color is yellow
The color cannot be determined

```

In networking many times we need to identify the class of an IPv4 address. The illustration below shows the range of addresses that belong to the different classes.

- Class A
1.0.0.0 – 126.0.0.0
- Class B
128.0.0.0 – 191.255.0.0

- Class C
192.0.0.0 – 223.255.255.0
- Class D
224.0.0.0 – 239.0.0.0
- Class E (Experimental)
240.0.0.0 – 248.0.0.0

The script below accepts an IP address from the command line and displays the class of the address. The class is determined by the first octet. The script first isolates the first octet of the address.

```
# script to determine the class of an IP address
param ( [string] $ipaddr )
$octet = $ipaddr.split(".") # Use the string split method to isolate the four octets
if ($octet[0] -lt 1) {
    write-host "$ipaddr is an invalid address"
} elseif ($octet[0] -le 126) {
    write-host "$ipaddr is a class A address"
} elseif ($octet[0] -eq 127) {
    write-host "$ipaddr is a loopback address"
} elseif ($octet[0] -le 191) {
    write-host "$ipaddr is a class B address"
} elseif ($octet[0] -le 223) {
    write-host "$ipaddr is a class C address"
} elseif ($octet[0] -le 239) {
    write-host "$ipaddr is a class D address"
} elseif ($octet[0] -le 240) {
    write-host "$ipaddr is experimental"
} else {
    write-host "$ipaddr is an invalid address"
}
```

```
1 # script to determine the class of an IP address
2 param ( [string] $ipaddr )
3 $octet = $ipaddr.split(".") # Use the string split method to isolate the four octets
4 if ($octet[0] -lt 1) {
5     write-host "$ipaddr is an invalid address"
6 } elseif ($octet[0] -le 126) {
7     write-host "$ipaddr is a class A address"
8 } elseif ($octet[0] -eq 127) {
9     write-host "$ipaddr is a loopback address"
10 } elseif ($octet[0] -le 191) {
11     write-host "$ipaddr is a class B address"
12 } elseif ($octet[0] -le 223) {
13     write-host "$ipaddr is a class C address"
14 } elseif ($octet[0] -le 239) {
15     write-host "$ipaddr is a class D address"
16 } elseif ($octet[0] -le 248) {
```

```
    write-host "$ipaddr is experimental"
18 } else {
19     write-host "$ipaddr is an invalid address"
20 }
```

```
PS C:\Users>
PS C:\Users> .\IPv4_class.ps1 '0.1.1.1'
0.1.1.1 is an invalid address

PS C:\Users> .\IPv4_class.ps1 '1.1.1.1'
1.1.1.1 is a class A address

PS C:\Users> .\IPv4_class.ps1 '123.1.1.1'
123.1.1.1 is a class A address

PS C:\Users> .\IPv4_class.ps1 '127.1.1.1'
127.1.1.1 is a loopback address

PS C:\Users> .\IPv4_class.ps1 '180.1.1.1'
180.1.1.1 is a class B address

PS C:\Users> .\IPv4_class.ps1 '193.1.1.1'
193.1.1.1 is a class C address

PS C:\Users> .\IPv4_class.ps1 '225.1.1.1'
225.1.1.1 is a class D address

PS C:\Users> .\IPv4_class.ps1 '238.1.1.1'
238.1.1.1 is a class D address

PS C:\Users> .\IPv4_class.ps1 '245.1.1.1'
245.1.1.1 is experimental

PS C:\Users> .\IPv4_class.ps1 '300.1.1.1'
300.1.1.1 is an invalid address
```

Switch Statement

Friday, November 13, 2015
10:57 AM

Writing an IF-ELSEIF statement when a number of condition need to be tested is very cumbersome and may result in code that is difficult to read. The SWITCH statement was developed for these cases.

```
switch -<options> (<value>) {  
    <pattern1> { <scriptblock1> }  
    <pattern1> { <scriptblock2> }  
    ...  
    <patternn> { <scriptblockn> }  
    default { <scriptblockm> }
```

The SWITCH statement functions in a similar manner as the IF-ELSEIF with the exception that the options make this statement much more powerful. The options may be -regex for regular expression matching, -wildcard for wildcard pattern matching, -exact for exact equality tests, -casesensitive for case sensitive matching, and -file for receiving the value from a file . The <value> may also be a pipeline that results in one or more values.

The one significant difference between switch and IF-ELSEIF is that in the switch statement the pattern may be matched multiple times.

The following are some simple examples of the use of the switch statement.

```
clear-host  
write-host  
$number = 4  
switch ($number) {  
    1 { write-host "The number is one" }  
    2 { write-host "The number is two" }  
    3 { write-host "The number is three" }  
    4 { write-host "The number is four" }  
    5 { write-host "The number is five" }  
    6 { write-host "The number is six" }  
    default { "Something else" }  
}  
write-host
```

```
1 clear-host
2 write-host
3 $number = 4
4 switch ($number) {
5     1 { write-host "The number is one" }
6     2 { write-host "The number is two" }
7     3 { write-host "The number is three" }
8     4 { write-host "The number is four" }
9     5 { write-host "The number is five" }
10    6 { write-host "The number is six" }
11    default { "Something else" }
12 }
13 write-host
14 |
```

```
<
```

```
The number is four
```

We modify the script to pipe a series of numbers into the switch statement to see the effect.

```
clear-host
write-host
5,3,10,2,5,8,6 | foreach-object {
    write-host "the current number is $_ `t" -nonewline
    switch ($_) {
        1 { write-host "The number is one" }
        2 { write-host "The number is two" }
        3 { write-host "The number is three" }
        4 { write-host "The number is four" }
        5 { write-host "The number is five" }
        6 { write-host "The number is six" }
        default { "Something else" }
    }
}
write-host
```

```

1  clear-host
2  write-host
3  5,3,10,2,5,8,6 | foreach-object {
4      write-host "the current number is $_ `t" -nonewline
5      switch ($_) {
6          1 { write-host "The number is one" }
7          2 { write-host "The number is two" }
8          3 { write-host "The number is three" }
9          4 { write-host "The number is four" }
10         5 { write-host "The number is five" }
11         6 { write-host "The number is six" }
12     default { "Something else" }
13 }
14 }
15 write-host

```

the current number is 5	The number is five
the current number is 3	The number is three
the current number is 10	Something else
the current number is 2	The number is two
the current number is 5	The number is five
the current number is 8	Something else
the current number is 6	The number is six

This next example is a reimplemention of the IF-ELSEIF example using the switch statement.

```

clear-host
write-host
'red', 'green', 'blue', 'cyan', 'yellow', 'orange' | foreach-object {
    switch ($_) {
        'red'   { write-host "The color is $_" -ForegroundColor $_ }
        'green' { write-host "The color is $_" -ForegroundColor $_ }
        'blue'  { write-host "The color is $_" -ForegroundColor $_ }
        'cyan'  { write-host "The color is $_" -ForegroundColor $_ }
        'yellow' { write-host "The color is $_" -ForegroundColor $_ }
        default { write-host "The color cannot be determined" -
ForegroundColor white}
    }
}
write-host

```

```

1  clear-host
2  write-host
3  'red', 'green', 'blue', 'cyan', 'yellow', 'orange' | foreach-object {
4    switch ($_) {
5      'red'   { write-host "The color is $_" -ForegroundColor $_ }
6      'green' { write-host "The color is $_" -ForegroundColor $_ }
7      'blue'  { write-host "The color is $_" -ForegroundColor $_ }
8      'cyan'  { write-host "The color is $_" -ForegroundColor $_ }
9      'yellow' { write-host "The color is $_" -ForegroundColor $_ }
10     default { write-host "THe color cannot be determined" -ForegroundColor white}
11   }
12 }
13 write-host

```

```

The color is red
The color is green
The color is blue
The color is cyan
The color is yellow
THe color cannot be determined

```

This script in the next example counts the number of dlls, exes, and other files in a given directory. In this script, we use a pipeline to pipe the file name to the switch statement. In the switch statement, the `-wildcard` option is used to indicated the match patterns contain wildcards. Note, this script could also be implemented with a simple pipeline using the group-object cmdlet. The only difference in the output is that group-object would enumerate every file type.

```

clear-host
write-host
$ndll = 0
$nexe = 0
$nother = 0
$dir = 'c:\windows\system32'
gci $dir | select-object name | % {
  switch -wildcard ($.name) {
    '*.dll' { $ndll++ }
    '*.exe' { $nexe++ }
    default { $nother++ }
  }
}
write-host "In the directory $dir there are:"

```

```
write-host "`t $ndll dlls, $nexe executables, and $nother other file types"
```

```
1  clear-host
2  write-host
3  $ndll = 0
4  $nexe = 0
5  $nother = 0
6  $dir = 'c:\windows\system32'
7  gci $dir | select-object name | % {
8    switch -wildcard ($_.name) {
9      '*.dll' { $ndll++ }
10     '*.exe' { $nexe++ }
11     default { $nother++ }
12   }
13 }
14 write-host "In the directory $dir there are:"
15 write-host "`t $ndll dlls, $nexe executables, and $nother other file types"
```

```
In the directory c:\windows\system32 there are:
2963 dlls, 547 executables, and 471 other file types
```

The next script is a rewrite of another script done in the IF-ELSEIF section using switch. This example demonstrates how bounds testing may occur on values. The script block representing the matching pattern must result in a true or false value. If the result is true the script block following is executed. Note, also the use of the *this* object metacharacter. In this case, it represents the value being tested.

```
# script to determine the class of an IP address
param ( [string] $ipaddr )
$octet = $ipaddr.split(".") # Use the string split method to isolate
                           the four octets
switch ($octet[0]) {
  {$_. -lt 1}                  { write-host "$ipaddr is an invalid
                                     address" }
```

```
{$_ -ge 1 -and $_ -le 126} { write-host "$ipaddr is a class A  
address" }  
{$_ -eq 127} { write-host "$ipaddr is a loopback  
address" }  
{$_ -ge 128 -and $_ -le 191} { write-host "$ipaddr is a class B  
address" }  
{$_ -ge 192 -and $_ -le 223} { write-host "$ipaddr is a class C  
address" }  
{$_ -ge 224 -and $_ -le 239} { write-host "$ipaddr is a class D  
address" }  
{$_ -ge 240 -and $_ -le 248} { write-host "$ipaddr is  
experimental" }  
default { write-host "$ipaddr is an invalid  
address" }  
}
```

```
1 # script to determine the class of an IP address
2 param ( [string] $ipaddr )
3 $octet = $ipaddr.split(".") # Use the string split method to isolate the four
4 $octet[0]) {
5     {$_ -lt 1} { write-host "$ipaddr is an invalid address" }
6     {$_ -ge 1 -and $_ -le 126} { write-host "$ipaddr is a class A address" }
7     {$_ -eq 127} { write-host "$ipaddr is a loopback address" }
8     {$_ -ge 128 -and $_ -le 191} { write-host "$ipaddr is a class B address" }
9     {$_ -ge 192 -and $_ -le 223} { write-host "$ipaddr is a class C address" }
10    {$_ -ge 224 -and $_ -le 239} { write-host "$ipaddr is a class D address" }
11    {$_ -ge 240 -and $_ -le 248} { write-host "$ipaddr is experimental" }
12    default { write-host "$ipaddr is an invalid address" }
13 }
```

```
PS C:\Users>
PS C:\Users> .\IPv4_class.ps1 0.1.1.1
0.1.1.1 is an invalid address

PS C:\Users> .\IPv4_class.ps1 1.1.1.1
1.1.1.1 is a class A address

PS C:\Users> .\IPv4_class.ps1 125.1.1.1
125.1.1.1 is a class A address

PS C:\Users> .\IPv4_class.ps1 127.1.1.1
127.1.1.1 is a loopback address

PS C:\Users> .\IPv4_class.ps1 130.1.1.1
130.1.1.1 is a class B address

PS C:\Users> .\IPv4_class.ps1 193.1.1.1
193.1.1.1 is a class C address

PS C:\Users> .\IPv4_class.ps1 225.1.1.1
225.1.1.1 is a class D address

PS C:\Users> .\IPv4_class.ps1 245.1.1.1
245.1.1.1 is experimental

PS C:\Users> .\IPv4_class.ps1 300.1.1.1
300.1.1.1 is an invalid address +
```

foreach

Tuesday, February 12, 2013
9:49 AM

The foreach *statement* operates in a similar fashion as the foreach-object *cmdlet*. The difference is that the foreach-object cmdlet must be used within a pipeline. The foreach statement is used to execute a scriptblock until the set objects specified is exhausted. Once the set of objects is exhausted or the set of objects is empty, execution of the scriptblock ceases and execution continues at the statement after the scriptblock.

```
foreach ( <variable> in <pipeline> ) {  
    <script block>  
}
```

In the above syntax, <pipeline> may be subexpression that returns a collection of objects or a collection of objects (array). <script block> many be any number of PowerShell statements. The <pipeline> may produce multiple objects. For each iteration of the script block, the <variable> holds the current object from the set of objects return by the <pipeline>. For example, for the first iteration, the <variable> holds the first object returned by the <pipeline>. For the second iteration, the <variable> holds the second object return by the <pipeline>. For last iteration, the <variable> holds the last object.

The following examples are taken from the foreach-object section and rewritten using the foreach statement. In the second example, the variable \$color is assigned the one value in the color array in sequence for each iteration of the loop. For the first iteration, the value \$color is "red". For the last iteration the value is 'blue'.

```
'red', 'green', 'blue' | foreach-object { write-host $_ }  
  
foreach ($color in 'red', 'green', 'blue' ) {  
    write-host $color  
}
```

```
PS C:\Users>
PS C:\Users> 'red', 'green', 'blue' | foreach-object { write-host $_ }
red
green
blue
PS C:\Users>
PS C:\Users> foreach ($color in 'red', 'green', 'blue') {
>>>   write-host $color
>>>
>>> }
red
green
blue
PS C:\Users>
```

```
'c:\windows\system', 'c:\windows\system32' |
foreach-object { get-childitem -path $_ -filter *.exe }

foreach ($dir in 'c:\windows\system', 'c:\windows\system32') {
  get-childitem -path $dir -filter *.exe
}
```

```
PS C:\Users>
PS C:\Users> 'c:\windows\system', 'c:\windows\system32' |
>>>   foreach-object { get-childitem -path $_ -filter *.exe }

Directory: C:\windows\system32

Mode                LastWriteTime       Length Name
----                -----        ---- 
-a---]      7/10/2015  6:59 AM        23040 acu.exe
-a---]      8/18/2015  3:04 AM      1234944 aitstatic.exe
-a---]      7/10/2015  6:59 AM        97792 alg.exe
-a---]      7/10/2015  7:00 AM        19456 appidcertstorecheck.exe
-a---]      7/10/2015  7:00 AM      161280 appidpolicyconverter.exe
-a---]      7/10/2015  7:00 AM        43416 ApplicationFrameHost.exe
-a---]      7/10/2015  6:59 AM        26112 ARP.EXE
-a---]      7/10/2015  7:00 AM        20505
```

```

PS C:\Users>
PS C:\Users> foreach ($dir in 'c:\windows\system', 'c:\windows\system32') {
>>>     get-childitem -path $dir -filter *.exe
>>>
>>> }

Directory: C:\windows\system32

Mode                LastWriteTime         Length Name
----                -----         ----- Name
-a---]    7/10/2015  6:59 AM           23040 acu.exe
-a---]    8/18/2015  3:04 AM          1234944 aitstatic.exe
-a---]    7/10/2015  6:59 AM           97792 alg.exe
-a---]    7/10/2015  7:00 AM          19456 appidcertstorecheck.exe
-a---]    7/10/2015  7:00 AM          161280 appidpolicyconverter.exe
-a---]    7/10/2015  7:00 AM           43416 ApplicationFrameHost.exe
-a---]    7/10/2015  6:59 AM           26112 ARP.EXE
-a---]    7/10/2015  7:00 AM           28696 at.exe

```

This next example enumerates a directory providing a file count and a total file size by type. The pipeline creates a set of fileInfo objects with the name and length of the corresponding file. The variable \$file is assigned each fileInfo object individually for each iteration of the script block.

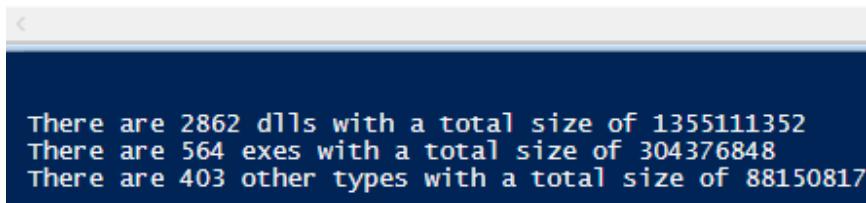
```

clear-host
write-host
$exeSize = $dllSize = $otherSize = 0
$exeCnt = $dllCnt = $otherCnt = 0
$dir = 'c:\windows\system32'
foreach ( $file in (gci $dir -file | select-object name, length) ) {
    switch -wildcard ($file.name) {
        '*.exe' {
            $exeSize += $file.length
            $exeCnt++
        }
        '*.dll' {
            $dllSize += $file.length
            $dllCnt++
        }
        default {
            $otherSize += $file.length
            $otherCnt++
        }
    }
}
write-host

```

```
write-host "There are $dllCnt dlls with a total size of $dllSize "
write-host "There are $exeCnt exes with a total size of $exeSize "
write-host "There are $otherCnt other types with a total size of
$otherSize "
```

```
1  clear-host
2  write-host
3  $exeSize = $dllsize = $otherSize = 0
4  $exeCnt = $dllcnt = $otherCnt = 0
5  $dir = 'c:\windows\system32'
6  foreach ( $file in (gci $dir -file | select-object name, length) ) {
7    switch -wildcard ($file.name) {
8      '*.exe' {
9        $exeSize += $file.length
10       $exeCnt++
11      }
12      '*.dll' {
13        $dllsize += $file.length
14        $dllcnt++
15      }
16      default {
17        $othersize += $file.length
18        $otherCnt++
19      }
20    }
21  }
22 }
23 write-host
24 write-host "There are $dllCnt dlls with a total size of $dllsize "
25 write-host "There are $exeCnt exes with a total size of $exeSize "
26 write-host "There are $otherCnt other types with a total size of $otherSize "
```



```
There are 2862 dlls with a total size of 1355111352
There are 564 exes with a total size of 304376848
There are 403 other types with a total size of 88150817
```

In this next script foreach is used to process a text file counting the number of words and lines in a file. The get-content cmdlet in the subexpression reads the complete text file into memory as a temporary string array. Each line in the file is contained in a separate entry in the array. The foreach sequentially processes each entry in the temporary array assigning it to the variable \$line.

The variable \$nlines is incremented for each iteration of the scriptblock. Since the scriptblock executes for each line in the file, we are essentially counting the lines in the file by incrementing \$nlines.

The split method of a string scans a string looking for a delimiter(s). Once it finds the delimiter, all text scanned up to the delimiter becomes a new string. This split continues scanning the string until all the text is exhausted. At that time, split returns all the strings as an array. In the example below, the split method is splitting the string using a space delimiter or a new-line delimiter (" `n"). Since each word in the current line of text is delimited by a space or a new-line (for the last word in the line), we are essentially splitting the line into words. The result of \$line.split(" `n") is a string array where each entry is one word. This array is assigned to the variable \$a. So, the length of the array is the number of words in the line. By adding \$a.length to \$nwords, we are accumulating the number of words in each line.

```
clear-host
write-host
$nlines = 0
$nwords = 0
$file = 'file.txt'
foreach ( $line in (get-content $file) ) {
    $nlines++                      # count the line
    $a = $line.split(" `n")          # split the line into a words store as an
array
    $nwords += $a.length            # the add the number of words to the total
}
write-host "The file $file has $nlines lines and $nwords total words"
Write-host
```

```
1  clear-host
2  write-host
3  $nlines = 0
4  $nwords = 0
5  $file = 'file.txt'
6  foreach ( $line in (get-content $file) ) {
7      $nlines++          # count the line
8      $a = $line.split(" `n") # split the line into a words store as an arry
9      $nwords += $a.length    # the addd the number of words to the total
10 }
11 write-host "The file $file has $nlines lines |and $nwords total words"
12 Write-host
```

```
The file file.txt has 7 lines and 95 total words
```

The example below is a sample script that runs chkdsk on a set of drives, stores the output of chkdsk to a text file. Critical to this script is that we must wait until each execution of checkdisk completes before gather the output of chkdsk and start the next instance. The scripts uses an execution string in the form of 'cmd / chkdsk d:' to execute chkdsk within a command shell. This is necessary to avoid issues when running chkdsk directly.

The syntax [System.Diagnostics.Process]::Start('cmd', "/c chkdsk.exe \$drive > \$chkdskout") passes to the Start method the process to execute, 'cmd', in the first parameter and command line parameters '/c (*the current drive*) > d:\mytemp\chkdskout.txt' in the second paramter. If the current drive is d: then the result would be:

```
[System.Diagnostics.Process]::Start('cmd', "/c chkdsk.exe d: >
d:\mytemp\chkdskout.txt")
```

Which results in execution the command shell with the command line command similar to that below for each drive listed in \$drives.

```
chkdsk.exe d: > d:\mytemp\chkdskout.txt
```

```
clear-host
write-host
$drives = 'b:','d:'
$chkdskall = 'd:\mytemp\chkdskall.txt'
remove-item $chkdskall
$chkdskout = 'd:\mytemp\chkdskout.txt'
foreach ($drive in $drives) {
    ('#' * 20) + " Checking $drive " + ('#' * 20) + "`n" | out-file
$chkdskall -append
    $p = [System.Diagnostics.Process]::Start('cmd', "/c chkdsk.exe
$drive > $chkdskout")
    wait-process -id $p.Id
    get-content $chkdskout | out-file $chkdskall -append
}
get-content $chkdskall # this retrieves the file and displays it on
the console
```

```

1 clear-host
2 write-host
3 $drives = 'b:','d:'
4 $chkdskall = 'd:\mytemp\chkdskall.txt'
5 remove-item $chkdskall
6 $chkdskout = 'd:\mytemp\chkdskout.txt'
7 foreach ($drive in $drives) {
8     ('#' * 20) + " Checking $drive " + ('#' * 20) + "`n" | out-file $chkdskall -append
9     $p = [System.Diagnostics.Process]::Start('cmd', "/c chkdsk.exe $drive > $chkdsko
10    wait-process -id $p.Id
11    get-content $chkdskout | out-file $chkdskall -append
12 }
13 get-content $chkdskall

```

```

#####
# Checking b: #####
The type of the file system is NTFS.
Volume label is RAID.

WARNING! F parameter not specified.
Running CHDKSK in read-only mode.

Stage 1: Examining basic file system structure ...
Progress: 0 of 336896 done; Stage: 0%; Total: 0%; ETA: 0:16:13
Progress: 38401 of 336896 done; Stage: 11%; Total: 3%; ETA: 0:15:35 .
Progress: 90550 of 336896 done; Stage: 26%; Total: 9%; ETA: 0:14:46 ..
Progress: 141002 of 336896 done; Stage: 41%; Total: 14%; ETA: 0:00:09 ...
Progress: 203265 of 336896 done; Stage: 60%; Total: 20%; ETA: 0:00:07
Progress: 262401 of 336896 done; Stage: 77%; Total: 26%; ETA: 0:00:07 .
Progress: 326146 of 336896 done; Stage: 96%; Total: 32%; ETA: 0:00:06 ..
Progress: 336896 of 336896 done; Stage: 100%; Total: 33%; ETA: 0:00:06 ...

336896 file records processed.

File verification completed.
Progress: 43 of 43 done; Stage: 100%; Total: 28%; ETA: 0:00:06

```

The script below pulls errors and warnings from selected sources from the event logs of selected servers. This script iterates through a list of servers. For each server it iterates through a list of logs retrieving errors and warnings from specific sources. All events found are save in a file in csv format .

This script uses a nested foreach statement. The outer foreach iterates through the server list. The inner foreach iterates through the logs list to be processed.

```
# add csv file header with a new line at the end
$msg = "Host,Log,Time Written, Event id,Source,Type`n"

$servers = 'host1','host2'
$logs = 'System', 'Application'
$logsrc = ' system','ESENT'

# Iterate thru all the servers
foreach ($server in $servers) {

    # iterate thru the logs for the server
    foreach ($log in $logs) {
        $events = get-eventlog -logname $log `

                            -computername $server -entrytype error, warning `

                            -source $logsrc `

                            -ErrorAction SilentlyContinue

        foreach ($event in $events)
        {
            $msg += "$server,"
            $msg += "$log,"
            $msg += "$($event.TimeWritten),"
            $msg += "$($event.EventID),"
            $msg += "$($event.Source),"
            $msg += "$($event.EntryType),"
            $msg += "`n"
        } # end events

    } # end logs

} # end servers

$msg | out-file log_entries.csv
```

While

Tuesday, February 12, 2013
11:05 AM

The while statement executes the scriptblock while the specified conditional expression evaluates to true. Once the expression evaluates to false, execution continues at the next statement after the scriptblock. The conditional expression is evaluated at the beginning of the loop, so if condition evaluates to false, the scriptblock is not executed. *Beware of infinite loops*, if the tested condition never evaluates to false, the execution of the scriptblock never terminates.

```
while( <conditional expression> ) {  
    <scriptblock>  
}
```

The examples below implement the same algorithms as the examples for the foreach statement.

This script displays the contents of the color array. Notice the difference between this script and the equivalent foreach script. The foreach statement is designed to handle collections of objects. The while is more general. To implement this script, we need to define an array index to index through the array of colors.

```
clear-host  
write-host  
$colors = 'red', 'green', 'blue'      # color array  
$i = 0  # the index into color array  
while  ( $i -lt ($colors.length - 1) ) {  
    write-host $colors[$i]  
    $i++  # increment the array index  
}  
write-host
```

```
1 clear-host
2 write-host
3 $colors = 'red', 'green', 'blue'      # color array
4 $i = 0 # the index into color array
5 while ( $i -lt $colors.length ) {
6     write-host $colors[$i]
7     $i++ # increment the array index
8 }
9 write-host
```

```
red
green
blue
```

This example is a rewrite of an example from the foreach statement section. The script counts files and totals file sizes by file type. In this example, we again see that the foreach statement is superior for handling collections. In this script, we have to collect all the fileInfo objects into the variable \$files. We then index into the \$files array in the body of the while script block.

```
clear-host
write-host
$exeSize = $dllSize = $otherSize = 0
$exeCnt = $dllCnt = $otherCnt = 0
$dir = 'c:\windows\system32'
$files = gci $dir -file | select-object name, length
$i = 0
while ( $i -lt $files.length ) {
    switch -wildcard ($files[$i].name) {
        '*.exe' {
            $exeSize += $file[$i].length
            $exeCnt++
        }
        '*.dll' {
            $dllSize += $file[$i].length
            $dllCnt++
        }
    }
    default {
        $otherSize += $file[$i].length
    }
}
```

```
        $otherCnt++
    }
}
$i++ # increment the array index
}
write-host
write-host "There are $dllCnt dlls with a total size of $dllSize "
write-host "There are $exeCnt exes with a total size of $exeSize "
write-host "There are $otherCnt other types with a total size of
$otherSize "
```

```

1  clear-host
2  write-host
3  $exeSize = $dllSize = $otherSize = 0
4  $exeCnt = $dllCnt = $otherCnt = 0
5  $dir = 'c:\windows\system32'
6  $files = gci $dir -file | select-object name, length
7  $i = 0
8  while ( $i -lt $files.length ) {
9    switch -wildcard ($files[$i].name) {
10      '*.exe' {
11        $exeSize += $files[$i].length
12        $exeCnt++
13      }
14      '*.dll' {
15        $dllSize += $files[$i].length
16        $dllCnt++
17      }
18    }
19    default {
20      $otherSize += $files[$i].length
21      $otherCnt++
22    }
23  }
24  $i++ # increment the array index
25 }
26 write-host
27 write-host "There are $dllCnt dlls with a total size of $dllSize "
28 write-host "There are $exeCnt exes with a total size of $exeSize "
29 write-host "There are $otherCnt other types with a total size of $otherSize "

```

```

There are 2963 dlls with a total size of 1607071160
There are 547 exes with a total size of 286307912
There are 353 other types with a total size of 70578222

```

In this example, the cmdlet get-content reads the file and creates an array where each entry contains one line in the file. This array is assigned to \$lines. The while loop executes until all entries in the \$lines array are processed. The expression

`$lines[$i].split(" `n")` splits the current entry in the `$lines` array using either a space or a new-line character as the *splitting* delimiter.

```
clear-host
write-host
$nwords = 0
$file = 'file.txt'
$lines = get-content $file
$i = 0
while ( $i -lt $lines.length ) {
    $a = $lines[$i].split(" `n")
    $nwords += $a.length
    $i++
}
write-host "The file $file has $($lines.length) lines and $nwords
total words"
write-host
```

```
1 clear-host
2 write-host
3 $nwords = 0
4 $file = 'file.txt'
5 $lines = get-content $file
6 $i = 0
7 while ( $i -lt $lines.length ) {
8     $a = $lines[$i].split(" `n")
9     $nwords += $a.length
10    $i++
11 }
12 write-host "The file $file has $($lines.length) lines and $nwords total words"
13 write-host|
```

```
The file file.txt has 7 lines and 95 total words
```

PowerShell supports a variation of the `while` statement called `do-while`. The `do-while` statement is similar to the `while` with the exception that the condition is tested at the end of the loop. This implies that the scriptblock is executed at least once regardless of the state of the condition that terminates the loop. The loop

iterates while the condition is true. Once the condition evaluates to false, the loop terminates and execution continues at the following statement.

```
do { <scriptblock> }
while( <conditional expression> )
```

The problems may be solved with while so the do-while has marginal value. In fact, due to the design of PowerShell many problems may be solved with foreach or the for statement.

For

Tuesday, February 12, 2013
1:29 PM

The for statement also called a counted loop executes a specific number of times. The loop statement uses an loop variable that is incremented as the loop executes. The loop terminates when the index meets some condition.

```
for ( <initialization>; <conditional expression>; <increment> ) {
    <scriptblock>
}
```

The purpose of <initialization> is to initialize the loop variable. <initialization> occurs only once before the script block is executed. As the name implies, <conditional expression> is a test for the condition that terminates the loop. Typically the condition is a test for when the loop variable reaches a certain value. The test for the condition occurs at the beginning of the scriptblock and occurs for each iteration of the scriptblock. The <increment> is a statement that typically increments the loop variable. The <increment> occurs at the end of the scriptblock each time the scriptblock executes.

The for loop has advantages over while and do-while when processing a collection. However, foreach is still superior in this case.

This first simple is a simple implementation of the for loop. The loop variable \$frequency is used in Beep method to specify the frequency of the sound.

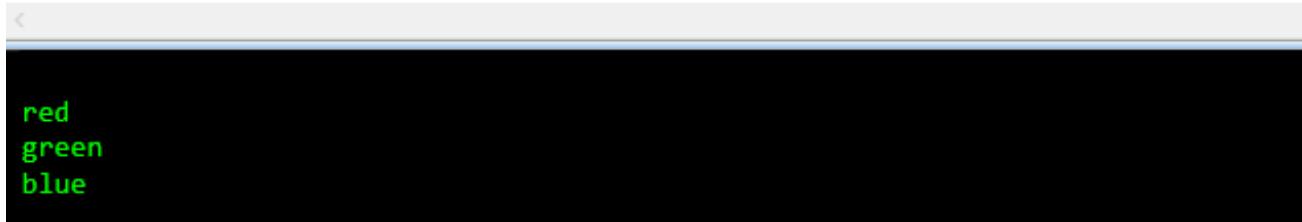
```
clear-host
write-host
# Output frequencies from 1000Hz to 4000Hz in 300Hz increments
for ($frequency=1000; $frequency -le 4000; $frequency +=300) {
    write-host "Current frequency $frequency"
    [System.Console]::Beep($frequency,100)
}
write-host
```

<http://powershell.com/cs/blogs/ebookv2/archive/2012/03/15/chapter-8-loops.aspx>

This script displays the contents of the color array and is based upon the similar script in the while statement section. Notice the difference between this script and the equivalent while script. The difference is essentially syntax. Just like the while statement, the for statement requires a loop index. The syntax allows the script developer to have the initialization, the loop termination test, and the increment of the loop index in one convenient place.

```
clear-host
write-host
$colors = 'red', 'green', 'blue'      # color array
for ($i = 0; $i -lt $colors.length; $i++) {
    write-host $colors[$i]
}
write-host
```

```
1 clear-host
2 write-host
3 $colors = 'red', 'green', 'blue'      # color array
4 for ($I = 0; $i -lt $colors.length; $I++) {
5     write-host $colors[$i]
6 }
7 write-host|
```



A screenshot of a terminal window showing the execution of a PowerShell script. The script defines an array of colors and loops through it, printing each color on a new line. The output is:

```
red
green
blue
```

This next script displays a pyramid of "*" on the console. It also illustrates the notion of a *nested loop*.

```
clear-host
write-host
for($i=1; $i -le 5; $i++) {
    for($j=0; $j -lt $i; $j++) {
        write-host '*' -NoNewline
    }
    write-host
}
```

```

1  clear-host
2  write-host
3  for($i=1; $i -le 5; $i++) {
4    for($j=0; $j -lt $i; $j++) {
5      write-host '*' -NoNewline
6    }
7  write-host
8 }

```

```

*
**
***
****
*****

```

Visuals are much better than numbers to show extremes. This script produces a bar chart of cpu usage. To scale that bar to something reasonable, which in this case 50 asterisks which limits the number of asterisks, we first find the maximum value in the set of all values. We then divide each cpu value by the max and multiple by 50. This script also use the -f format operator

<http://ss64.com/ps/syntax-f-operator.html>.

```

clear-host
write-host
$p = get-process | where-object { $_.cpu -gt 100 } | select name, cpu
# find the max cpu
$max = 0
for ($i=0; $i -lt $p.length; $i++) {
  if ($p[$i].cpu -gt $max) {
    $max = $p[$i].cpu
  }
}
for ($i=0; $i -lt $p.length; $i++) {
  $stars = ($p[$i].cpu / $max) * 50    # this gives me the cpu value
  scaled to 50
  "{0,20} {1,10:n0} {2,-50}" -f $p[$i].name, $p[$i].cpu, ('*' *
$stars)
}
write-host

```

```

1  clear-host
2  write-host
3  $p = get-process | where-object { $_.cpu -gt 100 } | select name, cpu
4  # find the max cpu
5  $max = 0
6  for ($i=0; $i -lt $p.length; $i++) {
7    if ($p[$i].cpu -gt $max) {
8      $max = $p[$i].cpu
9    }
10   }
11  for ($i=0; $i -lt $p.length; $i++) {
12    $stars = ($p[$i].cpu / $max) * 50    # this gives me the cpu value scaled to 50
13    "{0,20} {1,10:n0} {2,-50}" -f $p[$i].name,  $p[$i].cpu, ('*' * $stars)
14  }
15  write-host

```

The screenshot shows a command-line window displaying a list of processes and their CPU usage. The output is formatted with the process name on the left, its CPU usage in the middle, and a series of asterisks (*) representing the usage scale on the right. The processes listed include audiogd, chrome, explorer, g2mcomm, ipoint, MicrosoftEdgeCP, NvBackend, RuntimeBroker, SettingSyncHost, ShellExperienceHost, Snip, Solitaire, Steam, svchost, taskhostw, UA, and WinTVTray. The CPU usage values range from 101 to 16,367, and the asterisk scale varies accordingly.

Process Name	CPU Usage	Asterisk Scale
audiogd	6,014	*****
chrome	3,993	*****
chrome	1,413	****
chrome	2,128	*****
chrome	305	*
chrome	104	
chrome	1,944	*****
chrome	999	***
chrome	2,160	*****
chrome	553	**
chrome	643	**
chrome	332	*
explorer	289	*
g2mcomm	16,367	*****
ipoint	118	
MicrosoftEdgeCP	9,118	*****
MicrosoftEdgeCP	436	*
MicrosoftEdgeCP	11,392	*****
MicrosoftEdgeCP	17,654	*****
NvBackend	119	
RuntimeBroker	224	*
SettingSyncHost	312	*
ShellExperienceHost	1,225	***
Snip	760	**
Solitaire	1,808	*****
Steam	119	
svchost	198	*
taskhostw	101	
UA	109	
WinTVTray	338	*

This example is another rewrite of a script from the while statement section. Since the number of lines in the file is known after the file is read into memory. The for loop is more appropriate than the while.

```
clear-host
write-host
$nwords = 0
$file = 'file.txt'
$lines = get-content $file
for ($i = 0; $i -lt $lines.length; $i++) {
    $a = $lines[$i].split(" `n")
    $nwords += $a.length
}
write-host "The file $file has $($lines.length) lines and $nwords total words"
write-host
```

```
1 clear-host
2 write-host
3 $nwords = 0
4 $file = 'file.txt'
5 $lines = get-content $file
6 for ($i = 0; $i -lt $lines.length; $i++) {
7     $a = $lines[$i].split(" `n")
8     $nwords += $a.length
9 }
10 write-host "The file $file has $($lines.length) lines and $nwords total words"
11 write-host
```

```
The file file.txt has 7 lines and 95 total words
```

Loop Control

Tuesday, February 12, 2013
1:48 PM

Loop iteration is always terminated when a condition occurs. There are times when loop needs to be exited prematurely before the condition exists. The *break* statement provides this functionality. There are also times in the scriptblock of a loop when a condition arises that requires skipping the remainder of the statements in the scriptblock and proceeding to the next iteration of the loop. The *continue* statement provides this capability.

This example is taken from a previous section. The for loop is terminated by the break statement if the current color is green. So, the loop will terminate after the first entry in the \$colors array.

```
clear-host
write-host
$colors = 'red', 'green', 'blue', 'cyan', 'orange', 'purple'      #
color array
$i = 0 # the index into color array
while ( $i -lt $colors.length) {
    write-host "the loop index is $i and the color is $($colors[$i])"
    if ($colors[$i] -eq 'cyan') { break }
    $i++ # increment the array index
}
write-host
```

```
1 clear-host
2 write-host
3 $colors = 'red', 'green', 'blue', 'cyan', 'orange', 'purple'      # color array
4 $i = 0 # the index into color array
5 while ( $i -lt $colors.length) {
6     write-host "the loop index is $i and the color is $($colors[$i])"
7     if ($colors[$i] -eq 'cyan') { break }
8     $i++ # increment the array index
9 }
10 write-host|
```

<

```
the loop index is 0 and the color is red
the loop index is 1 and the color is green
the loop index is 2 and the color is blue
the loop index is 3 and the color is cyan
```

This next example demonstrates the use of continue. If the loop index is an even number, the control jumps to the end of the loop. In effect, this displays only the colors in odd numbered elements in the \$colors array. Note, the expression \$i % 2 produces the remainder in integer division. If the remainder is 0 the number is even divisible by 2, hence an even number. If the remainder is 1, the number is odd.

```
clear-host
write-host
$colors = 'red', 'green', 'blue', 'cyan', 'orange', 'purple'      #
color array
for ($i=0; $i -lt $colors.length; $i++) {
    if ( ($i % 2) -eq 0 ) { continue}
    write-host "the loop index is $i and the color is $($colors[$i])"
}
write-host
```

```
1 clear-host
2 write-host
3 $colors = 'red', 'green', 'blue', 'cyan', 'orange', 'purple'      # color array
4 for ($i=0; $i -lt $colors.length; $i++) {
5     if ( ($i % 2) -eq 0 ) { continue}
6     write-host "the loop index is $i and the color is $($colors[$i])"
7 }
8 write-host
```

```
<
the loop index is 1 and the color is green
the loop index is 3 and the color is cyan
the loop index is 5 and the color is purple
```

Basic Calculations

Tuesday, October 24, 2017

6:29 PM

The examples show below are taken from section "The Pipeline". These example for counting, totaling, find max, and finding the minimum are a rewrite of the examples found in that section. However, instead of usign a pipeline with Foreach-Object, they use the Foreach iteration statement.

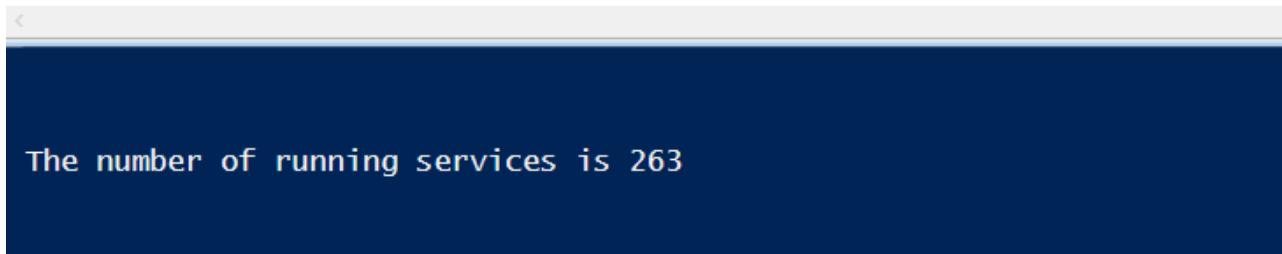
Counting the number of values in a set of values:

The algorithm for counting is very straightforward. Increment a variable every time the object passes through the pipeline.

This first example counts the number of services executing at the time the get-service cmdlet executes. Note that the counter \$nsvcs is initialized to 0 before the pipeline is executed.

```
clear-host
write-host `n`n
$nsvcs = 0
foreach ($service in (get-service)) {
    $nsvcs = $nsvcs + 1
}
write-host "The number of running services is $nsvcs"
write-host `n`n
```

```
1 clear-host
2 write-host `n`n
3 $nsvcs = 0
4 foreach ($service in (get-service)) {
5     $nsvcs++
6 }
7 write-host "The number of running services is $nsvcs"
8 write-host `n`n
```



```
The number of running services is 263
```

Generating a total of a set of values:

The algorithm for totaling (generating a sum or aggregating) a set of values is similar to that for counting but instead of incrementing a variable, we add the value, that we want to total, to the variable.

This example totals the free space on all disk drives in the computer.

```
clear-host
write-host `n`n
$totalfree = 0
foreach ($drive in (get-psdrive -psprovider filesystem)) {

    $totalfree += $drive.free
    # above syntax is the same as $totalfree = $totalfree + $drive.free
}

write-host "The amount of free space on all drives is $totalfree "
write-host `n`n
```

```

1 clear-host
2 write-host `n`n
3 $totalfree = 0
4 foreach ($drive in (get-psdrive -psprovider filesystem)) {
5
6     $totalfree += $drive.free
7     # above syntax is the same as $totalfree = $totalfree + $drive.free
8
9 }
10 write-host "The amount of free space on all drives is $totalfree "
11 write-host `n`n

```

The amount of free space on all drives is 53303341056

Finding the maximum value in set of values:

Finding the maximum value in a set of values requires a variable to hold the current maximum value. As objects pass through the pipeline, the current object value is tested against the variable value. If the current object value is greater than the variable value, the variable value is replaced by the current object value.

This example demonstrates the principle. The script determines the largest free space over all drives installed on the computer. Note that \$maxfree, the variable to hold the largest free space is initialized to 0. As the driveinfo objects flow through the pipeline, the free space of the current object is compared against the value of \$maxfree. If the free space of the current object is not greater than, where-object will drop the object from the pipeline and the foreach-object does not execute. If the free space of the current driveinfo object is greater than the value in \$maxfree, the object is passed on to foreach-object where the value of \$maxfree is replaced by the free space of the current drive info object.

```

clear-host
write-host `n`n
$maxfree = 0
foreach ($drive in (get-psdrive -psprovider filesystem)) {
    if ($drive.free -gt $maxfree) {
        $maxfree = $drive.free
    }
}
write-host "The largest free space is $maxfree"

```

```
    }
}

write-host "The largest free space on any drive is $maxfree "
write-host `n`n
```

```
1 clear-host
2 write-host `n`n
3 $maxfree = 0
4 foreach ($drive in (get-psdrive -psprovider filesystem)) {
5   if ($drive.free -gt $maxfree) {
6     $maxfree = $drive.free
7   }
8 }
9 write-host "The largest free space on any drive is $maxfree "
10 write-host `n`n
```

```
The largest free space on any drive is 32962994176
```

Finding the minimum value in a set of values:

Structurally the algorithm for finding the minimum is similar to find the maximum. The differences are the initial value assigned to the variable that holds the minimum value. The other difference is the comparison operator used. The algorithm requires that the current object value is compared against the variable holding the current minimum. If the current object value is less than the minimum variable value, then the minimum variable value is replaced by the current object value. Since the comparison operator is less than, the initial value assigned must be a large number.

The example below finds the smallest free space of all drives installed. Note the starting values assigned to \$minfree, 1000TB is a very large number. However, any number larger than the largest number in the set of values is sufficient.

```
clear-host
write-host `n`n
$minfree = 1000TB
foreach ($drive in (get-psdrive -psprovider filesystem)) {
```

```
if ($drive.free -lt $minfree) {  
    $minfree = $drive.free  
}  
}  
write-host "The smallest free space on any drive is $maxfree "  
write-host `n`n
```

```
1 clear-host  
2 write-host `n`n  
3 $minfree = 1000TB  
4 foreach ($drive in (get-psdrive -psprovider filesystem)) {  
5     if ($drive.free -lt $minfree) {  
6         $minfree = $drive.free  
7     }  
8 }  
9 write-host "The smallest free space on any drive is $maxfree "  
10 write-host `n`n
```

```
The smallest free space on any drive is 32962994176
```

Overview

Monday, February 18, 2013
10:06 AM

A function is an encapsulated script that may be executed repeatedly within a script by explicitly *calling* the function. Unlike scripts, functions reside in memory and must be loaded into the PowerShell address space before they may be executed.

A function may return a value to the calling environment. A function, regardless that it is embedded in a script, does not execute unless it is explicitly *called*. A function may also be loaded into the address space of the PowerShell console and used in the same manner as a cmdlet. Functions may also be added to the profile script and loaded into the address space when PowerShell starts, making them available from the command line.

```
function <function name> ( [<formal parameters>]) {  
    <function body>  
}
```

The <formal parameters> are optional function-local variables that exist only within the function once the function exits, these variables no longer exist. Formal parameters are distinguished from *actual parameters* which are present in the *calling* environment. The term that describes executing the function is *call*. This is a historical term that goes back to the early days of software development.

The <function body> is just a scriptblock that usually consists of multiple PowerShell statements.

Functions exist in memory and are only available for the session in which they were created. Functions present in scripts are available during the execution of the script.

The function below does not return a value but displays the time of day on the console.

```
function timeofday {  
    $d = (get-date).timeofday  
    $hr = $d.Hours  
    $min = $d.Minutes  
    $sec = $d.Seconds  
    "The time is {0,2:d2}:{1,2:d2}:{2,2:d2}" -f $hr, $min, $sec  
}
```

In this illustration below, the function is copied into the console window and is repeatedly executed.

```
PS C:\Users>
PS C:\Users> function timeofday {
>>>     $d = (get-date).timeofday
>>>     $hr = $d.Hours
>>>     $min = $d.Minutes
>>>     $sec = $d.Seconds
>>>     "The time is {0,2:d2}:{1,2:d2}:{2,2:d2}" -f $hr, $min, $sec
>>> }
PS C:\Users>
PS C:\Users> timeofday
The time is 08:55:24
PS C:\Users>
PS C:\Users> timeofday
The time is 08:55:30
PS C:\Users>
PS C:\Users> timeofday
The time is 08:55:33
PS C:\Users>
PS C:\Users> timeofday
The time is 08:55:37
PS C:\Users>
```

A better implementation would be to return a time string to the calling environment. In that way, the calling environment may decide what to do with the time string. The explicit typing of \$t in the function below is unnecessary since PowerShell would automatically type the variable to a string. The format string "{0,2:d2}:" means the first variable to the right of "-f", two display positions, and display two decimals and, if necessary, zero pad on the left, and followed by a ":".

```
function timeofday {
    $d = (get-date).timeofday
    $hr = $d.Hours
    $min = $d.Minutes
    $sec = $d.Seconds
    [string] $t = "{0,2:d2}:{1,2:d2}:{2,2:d2}" -f $hr, $min, $sec
    return $t
}
```

```

PS C:\Users>
PS C:\Users> function timeofday {
>>>     $d = (get-date).timeofday
>>>     $hr = $d.Hours
>>>     $min = $d.Minutes
>>>     $sec = $d.Seconds
>>>     [string] $t = "{0,2:d2}:{1,2:d2}:{2,2:d2}" -f $hr, $min, $sec
>>>     return $t
>>> }
PS C:\Users>
PS C:\Users> timeofday
08:59:00
PS C:\Users>
PS C:\Users> write-host "The time of day is $(timeofday)"
The time of day is 08:59:19

```

Let's create a new version of the function. This function receives a date string parameter and returns the time of day. This function verifies that the date string passed to it is a date. By using *try catch* if the conversion of the date string into a datetime type fails in the *try* script block, the *catch* script block executes. This example also illustrates the difference between formal and actual parameters, calling and called (function) environment. The parameter \$date is the formal parameter acting as a placeholder of the actual parameters. When the function is called in the ISE console below, the date strings on the command line are the actual parameters. The parameter value specified on the command line inhabits the formal parameter when the function is called.

```

function timeofday ( [string] $date ) {

    # if $date is not in the form of a date, we catch the error
    try { [datetime] $localdate = $date }
    catch {
        write-host "$date is not in the form of a date" -foregroundcolor
red
        return
    }

    $d = $localdate.timeofday
    $hr = $d.Hours
    $min = $d.Minutes
    $sec = $d.Seconds
    [string] $t = "{0,2:d2}:{1,2:d2}:{2,2:d2}" -f $hr, $min, $sec
    return $t
}

```

```

1 function timeofday ([string] $date) {
2
3     # if $date is not in the form of a date, we catch the error
4     try { [datetime] $localdate = $date }
5     catch {
6         write-host "$date is not a valid date" -foregroundcolor red
7         return
8     }
9
10    $d = $localdate.timeofday
11    $hr = $d.Hours
12    $min = $d.Minutes
13    $sec = $d.Seconds
14    [string] $t = "{0,2:d2}:{1,2:d2}:{2,2:d2}" -f $hr, $min, $sec
15    return $t
16 }
```

```

PS C:\Users>
PS C:\Users> timeofday '10/05/2015'
00:00:00

PS C:\Users> timeofday '10/05/2015 1:45:01 PM'
13:45:01

PS C:\Users> timeofday '1/05/2014 1:45:01 AM'
01:45:01

PS C:\Users> timeofday '12/32/2015 1:30:45 PM'
12/32/2015 1:30:45 PM is not a valid date
```

This example provides an additional examples of the difference between the formal and actual parameter. We also see how a function can test the existence of an actual parameter passed to it.

```

function hello ($name) {
    if ($name -eq $null) { # no name was passed
        write-host 'Hello World' -foreground magenta
    } else {
        write-host "Hello $name" -foreground yellow
    }
}
clear-host
write-host "`nCalling function hello`n" -ForegroundColor cyan
# this is the calling environment
hello          # calling the function without an actual parameter
hello 'Mike'    # calling the function with a string literal as an
actual parameter
```

```
$name = 'Jane'  
hello $name      # calling the function with the variable $name as  
an actual parameter  
write-host "`nend of script`n" -ForegroundColor cyan
```

Note: \$name in the calling environment is a different variable than \$name in the function.

```
1 function hello ($name) {  
2     if ($name -eq $null) {  
3         write-host 'Hello World' -foreground magenta  
4     } else {  
5         write-host "Hello $name" -foreground yellow  
6     }  
7 }  
8 clear-host  
9 write-host "`nCalling function hello`n" -ForegroundColor cyan  
10 # this is the calling environment  
11 hello          # calling the function without an actual parameter  
12 hello 'Mike'    # calling the function with a string literal as an actual parameter  
13  
14 $name = 'Jane'  
15 hello $name      # calling the function with the variable $name as an actual parameter  
16 write-host "`nend of script`n" -ForegroundColor cyan
```

```
calling function hello  
Hello World  
Hello Mike  
Hello Jane  
end of script
```

Typically functions return information back to the calling environment. This example uses a function that has no parameters and returns the OS version to the caller. The [environment] class is used to return the OS version and store it in \$osv. The function returns the major OS version to the caller.

```
PS C:\user>
PS C:\user> [environment]::osversion.version
```

Major	Minor	Build	Revision
-----	-----	-----	-----
6	1	7601	65536

```
function Get-OSVersion
{
    $osv = [environment]::OSVersion.version
    return $osv.major
}

get-osversion      # call the function
write-host "The OS version is $($get-osversion)"
switch (get-osversion) {
    10 { write-host "Windows 10 or Server 2016" }
    6.3 { write-host "Windows 8.1 or Server 2012 R2" }
    6.2 { write-host "Windows 8 or Server 2012" }
    6.1 { write-host "Windows 7 or Server 2008 R2" }
    default { write-host "You need to upgrade" }
}
```

```
1  function Get-OSVersion|
2  {
3      $osv = [environment]::OSVersion.version
4      return $osv.major
5  }
6
7  get-osversion      # call the function
8  write-host "The OS version is $($get-osversion)"
9  switch (get-osversion) {
10     10 { write-host "Windows 10 or Server 2016" }
11     6.3 { write-host "Windows 8.1 or Server 2012 R2" }
12     6.2 { write-host "Windows 8 or Server 2012" }
13     6.1 { write-host "Windows 7 or Server 2008 R2" }
14     default { write-host "You need to upgrade" }
15 }
```

Using \$args

Monday, February 17, 2014
7:57 AM

A function may accept parameters either through the use of formal parameters or by using the implicit parameter array \$args. Formal parameters should be used when the function uses a pre-determined number of parameters. Using formal parameters always makes a function more readable. However, there are times when the number of parameters passed to a function is not known beforehand, in this case the \$args array must be used.

```
function hello {
    write-host "Hello" $args[0] -foreground darkyellow
}
clear-host
write-host
hello          # call the function without a parameter
hello Bob      # call the function with a one string literal
hello Bob Carol # call the function with a two string literal
write-host
```

```
1 function hello {
2     write-host "Hello" $args[0] -foreground darkyellow
3 }
4 clear-host
5 write-host
6 hello          # call the function without a parameter
7 hello Bob      # call the function with a one string literal
8 hello Bob Carol # call the function with a two string literal
9 write-host
```



```
Hello
Hello Bob
Hello Bob
```

In line 6 of the script above, the function is called with an actual parameter. This causes the \$args array to be null. In the next line the function is called with the string 'Bob' which is stored as the first element of \$args in \$args[0]. In line 8, it's

called the function with two parameters. The string 'Bob' is stored in \$args[0] and 'Carol' is stored in \$args[1]. However the function only references \$args[0].

Let's re-write the function to accommodate multiple parameters. Instead of referencing \$args[0] in the write-host statement, we just specify the \$args array.

```
function hello {  
    write-host "Hello $args" -foreground darkyellow  
}  
clear-host  
write-host  
hello 'Bob'  
hello 'Bob' 'and' 'Carol'  
hello 'Bob' 'Carol' 'and' 'Ted'  
hello 'Bob' 'Carol' 'Ted' 'and' 'Alice'  
write-host
```

```
1 function hello {  
2     write-host "Hello $args" -foreground darkyellow  
3 }  
4 clear-host  
5 write-host  
6 hello 'Bob'  
7 hello 'Bob' 'and' 'Carol'  
8 hello 'Bob' 'Carol' 'and' 'Ted'  
9 hello 'Bob' 'Carol' 'Ted' 'and' 'Alice'  
10 write-host
```

```
Hello Bob  
Hello Bob and Carol  
Hello Bob Carol and Ted  
Hello Bob Carol Ted and Alice
```

This is the classic example of when \$args should be used. This function accepts a variable number of parameters. In the first invocation of the function, the string "Bob" is assigned to \$args[0]. When the write-host cmdlet is executed, PowerShell realizes that \$args has only one element so only \$args[0] is displayed. In the second call, the string "Bob" is assigned to \$args[0], "and" is assigned to \$args[1], and "Carol" assigned to \$args[2]. PowerShell recognizes that \$args has three elements and displays all three when write-host is executed. In the last call to hello, \$args has six elements containing: Bob, and, Carol, Ted, and, Alice.

What happens if some of the parameters are separated by commas:

```
function hello {  
    write-host "Hello $args" -foreground yellow  
}  
clear-host  
write-host  
hello Bob, Carol, Ted and Alice  
write-host
```

```
1 function hello {  
2     write-host "Hello $args" -foreground darkyellow  
3 }  
4 clear-host  
5 write-host  
6 hello Bob, Carol, Ted and Alice  
7  
8 write-host
```

```
Hello System.Object[] and Alice
```

What PS did in this case is pass three parameters to the function hello. A temporary array containing three elements: the string "Bob" in the first, the string "Carol" in the second, and "Ted" in the third. This temporary array was assigned to \$args[0], the second element of \$args contains "and", and \$args[2] contains the string "Alice".

```
function CheckArgs{  
    write-host "Arguments passed: " $args.length -foreground magenta  
    for ( $i=0; $i -lt $args.length; $i++ ) {  
        write-host "args[$i]: " $args[$i] -foreground darkyellow  
    }  
}  
clear-host  
write-host  
CheckArgs 'Bob'  
CheckArgs 'Bob' 'and' 'Carol'  
CheckArgs 'Bob' 'Carol' 'and' 'Ted'  
CheckArgs 'Bob' 'Carol' 'Ted' 'and' 'Alice'  
CheckArgs 1 3 5 7 9 11 13  
write-host
```

```
1 function CheckArgs{
2     write-host "Arguments passed: " $args.length -foreground magenta
3     for ( $i=0; $i -lt $args.length; $i++ ) {
4         write-host "args[$i]: " $args[$i] -foreground darkyellow
5     }
6 }
7 clear-host
8 write-host
9 CheckArgs 'Bob'
10 CheckArgs 'Bob' 'and' 'Carol'
11 CheckArgs 'Bob' 'Carol' 'and' 'Ted'
12 CheckArgs 'Bob' 'Carol' 'Ted' 'and' 'Alice'
13 CheckArgs 1 3 5 7 9 11 13
14 write-host
```

```
Arguments passed: 1
args[0]: Bob
Arguments passed: 3
args[0]: Bob
args[1]: and
args[2]: Carol
Arguments passed: 4
args[0]: Bob
args[1]: Carol
args[2]: and
args[3]: Ted
Arguments passed: 5
args[0]: Bob
args[1]: Carol
args[2]: Ted
args[3]: and
args[4]: Alice
Arguments passed: 7
args[0]: 1
args[1]: 3
args[2]: 5
args[3]: 7
args[4]: 9
args[5]: 11
args[6]: 13
```

This example demonstrates the flexibility of PowerShell. In this example, the function displays only the \$args elements that are populated. The for loop embedded in a subexpression in the display string returns only the populated values.

```
function hello {
```

```

    write-host "Hello $($for($i=0; $i -lt $args.length; $i++) {
$args[$i]} )" `

                                         -foreground darkyellow
}
clear-host
write-host
hello 'Bob'
hello 'Bob' 'and' 'Carol'
hello 'Bob' 'Carol' 'and' 'Ted'
hello 'Bob' 'Carol' 'Ted' 'and' 'Alice'
write-host

```

```

Hello Bob
Hello Bob and Carol
Hello Bob Carol and Ted
Hello Bob Carol Ted and Alice

```

This is a more practical example where the function retrieves entries from an event log. The first parameter specified is the event log name, parameters following are event log sources.

```

function get-elogs {

# check to make sure parameters were passed
if ( $args[0] -eq $null ) {
    write-host "need to pass log name (s) and source(s)" -foreground red
    return
}

# get the log(s) name
$logs = $args[0]

# create an empty array
$sources = @()
# this loops thru the command line parameters starting with the second
for ( $i=1; $i -lt $args.length; $i++ ) {
    # append the command line parameter
    $sources += $args[$i]
}

# the -source parameter supports arrays
get-eventlog -logname $logs -source $sources | format-table -autosize
}
```

```

>>> }
>>>
>>> # get the log(s) name
>>> $logs = $args[0]
>>>
>>> # create an empty array
>>> $sources = @()
>>> # this loops thru the command line parameters starting with the second
>>> for ( $i=1; $i -lt $args.Length; $i++ ) {
>>>     # append the command line parameter
>>>     $sources += $args[$i]
>>> }
>>>
>>> # the -source parameter supports arrays
>>> get-eventlog -logname $logs -source $sources | format-table -autosize
>>>
>>> }
PS C:\Users>
PS C:\Users> get-elogs
need to pass log name (s) and source(s)
PS C:\Users>
PS C:\Users> get-elogs 'system' 'user32'

Index Time           EntryType   Source InstanceID Message
---- --           -----   -----
 635 Sep 23 16:12 Information User32 2147484722 The process C:\WINDOWS\Explorer.EXE (IS
 496 Sep 22 18:07 Information User32 2147484722 The process C:\WINDOWS\Explorer.EXE (IS
 275 Sep 22 16:53 Information User32 2147484722 The process C:\WINDOWS\system32\winlogon

PS C:\Users> get-elogs 'system' 'user32' 'Microsoft-Windows-Dhcp-client'

Index Time           EntryType   Source           InstanceID Message
---- --           -----   -----
 635 Sep 23 16:12 Information User32           2147484722 The process C:\W
 615 Sep 23 15:56 Information Microsoft-Windows-Dhcp-Client 50036 DHCPv4 client se
 579 Sep 23 15:55 Information Microsoft-Windows-Dhcp-Client 50037 DHCPv4 client se
 496 Sep 22 18:07 Information User32           2147484722 The process C:\W
 313 Sep 22 16:57 Information Microsoft-Windows-Dhcp-Client 50036 DHCPv4 client se
 278 Sep 22 16:53 Information Microsoft-Windows-Dhcp-Client 50037 DHCPv4 client se
 275 Sep 22 16:53 Information User32           2147484722 The process C:\W
 35 Sep 22 16:40 Information Microsoft-Windows-Dhcp-Client 50036 DHCPv4 client se

```

Using Formal Parameters

Monday, February 18, 2013

11:38 AM

Formal parameters are placeholders that identify to PowerShell the number and type of parameters passed. If a type isn't specified for a *formal parameter*, PowerShell will attempt to determine the type when the *actual parameter* is passed.

```

function <function name> ( <parameter1>, <parameter2>, ... ) {
    <function body>
}

```

Where <parameter1> ... are the optional formal parameters.

The function returns the pathname for a file whose name is passed in the formal parameter \$filename. The function searching the root of the volume found in the other formal parameter \$drive. The function first validates the drive letter. If the value in \$drive is invalid, the function displays an error message and returns null to the calling environment. The function use a foreach loop to search all the fileInfo objects return by get-childitem for the drive. If the file name in the formal parameter \$filename matches the value of name property of the fileInfo object. The value of the directoryname property is store in \$foundpath and the loop is ended using the break statement.

```

# This function searches a drive for a file and returns the pathname
where a file is found
function find-file ( [char] $drive, [string] $filename ) {

    [char[]] $drives = 'a','b','c','d','e','f','g','h','i','j' ` 
                      , 'k','l','m','n','o','p','q','r','s','t' ` 
                      , 'u','v','w','x','y','z'

    # is this a valid drive letter
    if ($drive -notin $drives) {
        write-host 'invalid drive letter $drive' -foreground red
        return
    }

    $foundpath = $null
    $mypath = $drive + ":\" 
    foreach ($file in ( gci -path $mypath -recurse -file -ErrorAction
SilentlyContinue ) ) {

        if ( $filename -eq $file.name ) {
            $foundpath = $file.directoryname
            break
        }
    }
}

```

```
    return $foundpath  
}  
}
```

We may now call the function with a *positional actual parameters* as below:

```
1 # This function searches a drive for a file and returns the pathname where a file is f  
2 function find-file ([char] $drive, [string] $filename) {  
3     [char[]] $drives = 'a','b','c','d','e','f','g','h','i','j' `  
4             , 'k','l','m','n','o','p','q','r','s','t' `  
5             , 'u','v','w','x','y','z'  
6  
7     # is this a valid drive letter  
8     if ($drive -notin $drives) {  
9         write-host 'invalid drive letter' -foreground red  
10        return  
11    }  
12  
13    $foundpath = $null  
14    $mypath = $drive + ":"\  
15    foreach ($file in ( gci -path $mypath -recurse -ErrorAction SilentlyContinue ) ) {  
16        if ( $filename -eq $file.name ) {  
17            $foundpath = $file.directoryname  
18            break  
19        }  
20    }  
21  
22    }  
23    return $foundpath  
24 }  
25 }  
26 }  
27 }  
28 }  
  
PS C:\Users>  
PS C:\Users> find-file 'c' 'hosts'  
C:\windows\System32\drivers\etc
```

PowerShell provides us with syntax to make the function call more readable. In the example below the function is called with *named actual parameters*. Also shown below are the errors that may occur when the \$drive parameter is not valid. Notice that in the last example, \$null is returned to the calling environment since the file was not found.

```

PS C:\Users>
PS C:\Users> # This function searches a drive for a file and returns the pathname where a file is
PS C:\Users> function find-file ( [char] $drive, [string] $filename ) {
>>>
>>>     [char[]] $drives = 'a','b','c','d','e','f','g','h','i','j' :
>>>             , 'k','l','m','n','o','p','q','r','s','t' :
>>>             , 'u','v','w','x','y','z'
>>>
>>>     # is this a valid drive letter
>>>     if ($drive -notin $drives) {
>>>         write-host 'invalid drive letter' -foreground red
>>>         return
>>>     }
>>>
>>>     $foundpath = $null
>>>     $mypath = $drive + ":\" 
>>>     foreach ($file in ( gci -path $mypath -recurse -ErrorAction SilentlyContinue ) ) {
>>>
>>>         if ( $filename -eq $file.name ) {
>>>             $foundpath = $file.directoryname
>>>             break
>>>         }
>>>
>>>     }
>>>     return $foundpath
>>>
>>> }
PS C:\Users>
PS C:\Users> find-file -drive 'c' -filename 'hosts'
C:\Windows\System32\drivers\etc
PS C:\Users>
PS C:\Users> find-file -filename 'hosts'
invalid drive letter
PS C:\Users>
PS C:\Users> find-file -drive '1' -filename 'hosts'
invalid drive letter
PS C:\Users>
PS C:\Users> find-file -drive 'c' -filename 'doesntexist'
PS C:\Users>

```

Typed format Parameters:

Formal parameters may be typed to eliminate the need for dynamic interpretation of the type. Typing also provides built-in error checking to minimize unpredictable results. Finally typing formal parameters provides documentation as to what the function is expecting.

The value of typing is illustrated in the example below. The purpose of this function is to add two integers together. However, since the "+" operator is overload for numbers and strings this function would return unexpected results.

```

# add two integers
function add-two ( $i , $j ) {

    return $i + $j
}

```

As we can see below, passing two strings to the function returned a concatenated string. Ideally, we want to avoid this effect. In the first execution, no parameters are passed so the formal parameters have a value of null. The function returns \$null + \$null which is \$null. In the second execution the function performed as advertised. In the third, the addition was completed by a floating point not an integer was return. The fourth execution the concatenation of the string "5" and "7". This occurred since for strings the "+" operator produces this result. Ideally, we would like PowerShell to cast these parameters to integers. However, PowerShell has not information as to how to cast. The final execution is even more problematic since these strings should have caused an error.

```
PS C:\Users>
PS C:\Users> # add two integers
PS C:\Users> function add-two ( $i , $j ) {
>>>
>>>     return $i + $j
>>> }
PS C:\Users>
PS C:\Users> add-two
PS C:\Users>
PS C:\Users> add-two 5 7
12
PS C:\Users> add-two 5.1 7.3
12.4
PS C:\Users> add-two '5' '7'
57
PS C:\Users> add-two 'a' 'b'
ab
```

To correct the above effects, the function header is re-written so that the parameters are typed.

```
# add two integers
function add-two ( [int] $i , [int] $j ) {

    return $i + $j
}
```

Notice the difference in the function execution. In the first execution, PowerShell cast the \$null to an 0 integer so an integer result was returned. In the third

execution, the floating point numbers were truncated, cast to integers, and the addition completed. In the third execution, PowerShell was able to cast the two strings into integers. The last execution example clearly demonstrates the benefit of type. In the case an error was thrown since the strings 'a' and 'b' cannot be cast into integers.

```
PS C:\Users>
PS C:\Users> # add two integers
PS C:\Users> function add-two ( [int] $i , [int] $j ) {
>>>
>>>     return $i + $j
>>>
PS C:\Users>
PS C:\Users> add-two
0
PS C:\Users> add-two 5 7
12
PS C:\Users> add-two 5.1 7.3
12
PS C:\Users> add-two '5' '7'
12
PS C:\Users> add-two 'a' 'b'
add-two : Cannot process argument transformation on parameter 'i'. Cannot
Error: "Input string was not in a correct format."
At line:1 char:9
+ add-two 'a' 'b'
+
+ CategoryInfo          : InvalidData: (:) [add-two], ParameterBindin
+ FullyQualifiedErrorId : ParameterArgumentTransformationError,add-
```

Default values for formal parameters:

Default values for formal parameters provide flexibility in writing functions since the caller doesn't have to specify all specified parameters.

```
function <function name> ( <parameter1> = <expr1>, <parameter2> = <expr2>,
... ) {
    <function body>
}
```

This example is taken from the previous section. The formal parameters have default values. If this function is called with no actual parameters, the formal parameters are assigned the default values.

```
# add two integers
function add-two ( [int] $i=1 , [int] $j=2 ) {
    return $i + $j
}
```

```
PS C:\user>
PS C:\user> # add two integers
PS C:\user> function add-two ( [int] $i=1 , [int] $j=2 ) {
>>>
>>>     return $i + $j
>>>
PS C:\user>
PS C:\user> add-two
3
PS C:\user> add-two 5 7
12
```

Sub-expressions may be used in generating the default value. In the example below, the default value is set by get-date.

```
function day-of-week ( [datetime] $d = (get-date) ) {
    $d.dayofweek
}
```

```
PS C:\user>
PS C:\user> function day-of-week ( [datetime] $d = (get-date) ) {
>>>     $d.dayofweek
>>>
PS C:\user>
PS C:\user> day-of-week 'January 23, 1949'
Sunday
PS C:\user>
PS C:\user> day-of-week 'December 7, 1941'
Sunday
PS C:\user> get-date
Thursday, October 08, 2015 1:53:21 PM

PS C:\user> day-of-week
Thursday
PS C:\user>
```

In this example, if an IP address is not passed on the command line the user is prompted to enter it.

```
# an example of splatting, i.e., passing a hash table to a cmdlet
function test-host ( $ip = (read-host "Enter an IP address")) {
    $pingOptions = @{"count" = 1;"bufferSize" = 15;"delay" = 1}
    test-connection $ip @pingOptions
}
```

```
PS C:\user>
PS C:\user> # an example of splatting, i.e., passing a hash table to a cmdlet
PS C:\user> function test-host ( $ip = $(read-host "Enter an IP address")) {
>>>     $pingOptions = @{"count" = 1;"bufferSize" = 15;"delay" = 1}
>>>     test-connection $ip @pingOptions
>>> }
PS C:\user>
PS C:\user> test-host 8.8.8.8
Source      Destination      IPV4Address      IPV6Address      Bytes      Ti
-----      -----      -----      -----
MINK        8.8.8.8          8.8.8.8          -----          15          45

PS C:\user> test-host
Enter an IP address: 8.8.4.4
Source      Destination      IPV4Address      IPV6Address      Bytes      Ti
-----      -----      -----      -----
MINK        8.8.4.4          8.8.4.4          -----          15          45
```

Returning values

Tuesday, February 19, 2013

4:48 PM

Any output generated within a function is returned back to the calling environment. The examples below demonstrates this concept.

This first example demonstrates the difference between putting a standalone variable in the main block of the script versus doing the same within a function. In this example, the variable \$str stands alone in the main block which causes the contents of the variable to be displayed on the console. Inside the function the variable \$str, which is a different variable than the one in the main block, stands alone. In this case, the contents of \$str are returned to the calling environment. This is an example the function implicitly returning a value by having the variable \$str standalone.

```

function myfunc {
    $str = 'wxyz'      # this is not the same variable as $str in the
main block
    $str # this is not displayed but returned to the calling
environment
}

$str = 'abcdef'
$str # this displays in the console window

$retval = myfunc # call the function

$str
$retval

```

```

PS C:\Users>
PS C:\Users> function myfunc {
>>>     $str = 'wxyz'      # this is not the same variable as $str in the main blo
>>>     $str # this is not displayed but return to the calling environment
>>> }
PS C:\Users>
PS C:\Users>
PS C:\Users> $str = 'abcdef'
PS C:\Users> $str # this displays in the console window
abcdef
PS C:\Users>
PS C:\Users> $retval = myfunc # call the function
PS C:\Users>
PS C:\Users> $str
abcdef
PS C:\Users>
PS C:\Users> $retval
wxyz
PS C:\Users>

```

What happens when I have multiple standalone variables in a function? In this case, as shown below, when the function exits an array containing the values of the standalone variables is returned to the caller. The function multi-values returns an array containing the three strings standing alone inside the function.

```

function multi-values {
    'first value'
    'second value'
    'third value'
}

```

```
$retval = multi-values

$retval.GetType()

$retval

$retval | gm | select -first 5
```

```
PS C:\Users>
PS C:\Users> function multi-values {
>>>     'first value'
>>>     'second value'
>>>     'third value'
>>> }
PS C:\Users>
PS C:\Users> $retval = multi-values
PS C:\Users>
PS C:\Users> $retval.GetType()

IsPublic IsSerial Name                                     BaseType
----- ----- -- Object[]                                System.Array

PS C:\Users> $retval
first value
second value
third value
PS C:\Users>
PS C:\Users> $retval | gm | select -first 5

    TypeName: System.String

Name      MemberType Definition
----      -----
Clone    Method     System.Object Clone(), System.Object ICloneable.
CompareTo Method   int CompareTo(System.Object value), int CompareTo(
Contains  Method   bool Contains(string value)
CopyTo   Method    void CopyTo(int sourceIndex, char[] destination,
EndsWith Method   bool EndsWith(string value), bool EndsWith(strin
```

The function below upcases the string passed to it. The function assigns the upcased string to \$newstr within the function and does not return a value to the

calling environment. When the function is called from the main block of the script, since the function returns no value, the value of the variable \$newstr is null. Note, that the formal parameter \$str is typed as a [string]. Doing this allows us to use the ToUpper() method of the string class to upcase the string. What is also important to note is that the variables \$newstr defined in the function is local to the function and is a different than \$newstr in the main block. Once we exit the function any local variables cease to exist.

```
function Upcase ([string] $str) {  
  
    $newstr = $str.ToUpper()  
}  
  
$newstr = upcase 'hello'  
write-host "The value of newstr is $newstr" -ForegroundColor magenta  
write-host "The value of returned is $(Upcase 'hello')" -  
ForegroundColor yellow
```

```
PS C:\Users>  
PS C:\Users> function Upcase ([string] $str) {  
    >>>  
    >>>     $newstr = $str.ToUpper()  
    >>> }  
PS C:\Users>  
PS C:\Users> $newstr = upcase 'hello'  
PS C:\Users> write-host "The value of newstr is $newstr" -ForegroundColor magenta  
The value of newstr is  
PS C:\Users> write-host "The value of returned is $(Upcase 'hello')" -ForegroundColor yellow  
The value of returned is
```

To correct this behavior the function must return a value to the calling environment. This may be done implicitly or explicitly. The example below illustrates the implicit return. However the return of control to the calling environment occurs because there are no more statements in the function and not because of the standalone variable as seen earlier in this section.

```
function Upcase ([string] $str) {  
  
    # this implicitly returns the upcased string  
    $str.ToUpper()  
}  
  
$newstr = upcase 'hello'  
write-host "The value of newstr is $newstr" -ForegroundColor magenta
```

```
write-host "The value of returned is $(Upcase 'hello')" -  
ForegroundColor yellow
```

```
PS C:\Users>  
PS C:\Users> function Upcase ([string] $str) {  
>>>     # this implicitly returns the upcased string  
>>>     $str.ToUpper()  
>>> }  
PS C:\Users>  
PS C:\Users> $newstr = upcase 'hello'  
PS C:\Users> write-host "The value of newstr is $newstr" -ForegroundColor magenta  
The value of newstr is HELLO  
PS C:\Users> write-host "The value of returned is $(Upcase 'hello')" -ForegroundColor Co  
The value of returned is HELLO
```

Having an implicit return as shown above *is not a best practice* because there is no clear declaration of what the function is attempting to do. For clarity of intent as to what is the value returned to the calling environment, the *return* statement should be used.

The return statement functions similar to the break but optionally allows a value (object) to be returned to the calling environment. When used within a function, return causes execution of the function to end and control returned back to the calling environment.

```
return  
return <expression>
```

The previous example is rewritten to use the return statement to end execution of the function and returned the value \$newstr to the calling environment.

```
function Upcase ([string] $str) {  
  
    $newstr = $str.ToUpper()  
    return $newstr  
}  
  
$newstr = upcase 'hello'  
write-host "The value of newstr is $newstr" -ForegroundColor magenta
```

```
write-host "The value of returned is $(Upcase 'hello')" -  
ForegroundColor yellow
```

```
PS C:\Users>  
PS C:\Users> function Upcase ([string] $str) {  
>>>  
>>>    $newstr = $str.ToUpper()  
>>>    return $newstr  
>>> }  
PS C:\Users>  
PS C:\Users> $newstr = upcase 'hello'  
PS C:\Users> write-host "The value of newstr is $newstr" -ForegroundColor magenta  
The value of newstr is HELLO  
PS C:\Users>  
PS C:\Users> write-host "The value of returned is $(Upcase 'hello')" -ForegroundColor yellow  
The value of returned is HELLO
```

Let's look at a more practical example. The script below accepts an IP address as a parameter and returns the string "alive" or "dead" depending on response or lack of response from the target host.

```
function ping-host ( [string] $ip = $($() throw "An IP address is  
required" )) {  
  
    $IsValidIPaddress = $ip -match '\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}'  
    If (-not $IsValidIPaddress) {  
        write-host "$ip is not a valid ip address" -ForegroundColor red  
        return 'IP address is not valid'  
    }  
  
    $r = test-connection $ip -count 1 -BufferSize 15 -ErrorAction  
    SilentlyContinue  
    if ($r.StatusCode -eq 0) {  
        return 'alive'  
    }  
  
    return 'dead'  
}  
  
# invalid ip  
$h = 'Not an ip address'  
$r = ping-host $h  
write-host "Host $h is $r" -ForegroundColor Yellow
```

```
# invalid ip
$h = '999.999.999.999'
$r = ping-host $h
write-host "Host $h is $r" -ForegroundColor Yellow

# non-existent ip
$h = '172.16.1.2'
$r = ping-host $h
write-host "Host $h is $r" -ForegroundColor Yellow

# valid ip
$h = '198.111.176.6'
$r = ping-host $h
write-host "Host $h is $r" -ForegroundColor Yellow
```

```

PS C:\Users>
PS C:\Users> function ping-host ([string] $ip = $($ throw "An IP address is required"))
>>>     $isValidIPAddress = $ip -match '\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}'
>>>     If (-not $isValidIPAddress) {
>>>         write-host "$ip is not a valid ip address" -ForegroundColor red
>>>         return 'IP address is not valid'
>>>     }
>>>
>>>     $r = test-connection $ip -count 1 -BufferSize 15 -ErrorAction SilentlyContinue
>>>     if ($r.StatusCode -eq 0) {
>>>         return 'alive'
>>>     }
>>>
>>>     return 'dead'
>>> }

PS C:\Users>
PS C:\Users> # invalid ip
PS C:\Users> $h = 'Not an ip address'
PS C:\Users> $r = ping-host $h
Not an ip address is not a valid ip address
PS C:\Users> write-host "Host $h is $r" -ForegroundColor Yellow
Host Not an ip address is IP address is not valid
PS C:\Users>
PS C:\Users> # invalid ip
PS C:\Users> $h = '999.999.999.999'
PS C:\Users> $r = ping-host $h
PS C:\Users> write-host "Host $h is $r" -ForegroundColor Yellow
Host 999.999.999.999 is dead
PS C:\Users>
PS C:\Users> # non-existent ip
PS C:\Users> $h = '172.16.1.2'
PS C:\Users> $r = ping-host $h
PS C:\Users> write-host "Host $h is $r" -ForegroundColor Yellow
Host 172.16.1.2 is dead
PS C:\Users>
PS C:\Users> # valid ip
PS C:\Users> $h = '198.111.176.6'
PS C:\Users> $r = ping-host $h
PS C:\Users> write-host "Host $h is $r" -ForegroundColor Yellow
Host 198.111.176.6 is dead

```

Let's look at the error checking in this function. The first check is specified in the parameter syntax.

```
[string] $ip = $($ throw "An IP address is required" ))
```

If no parameter is passed to the function and error is thrown with the message "An IP address is required". If this happens, the function does not execute and control is returned to the calling environment.

If next check validates that the function received something that looks like an IP address. The parameter is *matched* against the regular expression.

```
$IsValidIPaddress = $ip -match  
'\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}'
```

The above line in the function matches the IP address string to a regular expression. We attempt to verify the format of the IP address and not the content. The regular expression matches three digits followed by a "." followed by the same match pattern three more times. The expression "\d{1,3}\." is interpreted as follows "\d" - decimal number, {1,3} - between one and three digits in length, and "\." followed by a ".". Note, the "\" is an escape character. The escape character means not to interpret the character in the normal sense. So "\d" means the "d" must be interpreted as the metacharacter indicating a decimal digit instead of just the character "d". In regular expressions the character "." is a metacharacter that means match any character. When escaped as in the string "\." the "." is interpreted as a regular character.

Note when the function was passed the IP address 999.999.999.999 no error was thrown and the ping was attempted. The regular expression test for the format of an IP address, i.e., 1-3 digits followed by a "." repeated four times. It is possible to validate content of an IP address using a regular expression. However, it is much more complex.

The function then executes use the test-connection cmdlet to ping the function and return the appropriate status to the calling environment based upon the property StatusCode of the response.

Switch Parameters

Tuesday, February 19, 2013
1:55 PM

Switch parameters are similar to cmdlet switches. A cmdlet switch has a name (like other parameters) but no value associated with the name. A switch parameter functions like a *boolean*, i.e., has a value of true or false. If the switch

parameter is specified the value assigned to it is true otherwise the value assigned is false.

For a parameter to be considered a switch parameter, it must be typed as [switch].

```
function <function name> ( [switch] <parameter1>, ...) {  
    <function body>  
}
```

The example below shows how a switch parameter may be used to alter the script execution within the function. The function below accepts two parameters: a string and a switch parameter. If the switch parameter is specified the function upper cases the string otherwise the original string is returned.

```
function ApplyCase ([switch] $upper, [string] $str) {  
  
    if ( $upper ) {  
        $newstr = $str.ToUpper()  
    } else {  
        $newstr = $str.ToLower()  
    }  
    return $newstr  
}  
  
clear-host  
write-host  
$actualstring = 'THE QUICK BROWN FOX JUMPS OF THE LAZY DOG'  
  
write-host "Function call with no switch parameter" -foreground yellow  
$returnedstring = ApplyCase $actualstring  
write-host "`tThe function ApplyCase: `n`t`t was called with  
$actualstring `n`t`t and returned $returnedstring" -ForegroundColor magenta  
write-host  
write-host "Function call with -upper parameter" -foreground yellow  
$actualstring = $returnedstring  
$returnedstring = ApplyCase $actualstring -upper  
write-host "`tThe function ApplyCase: `n`t`t was called with  
$actualstring `n`t`t and returned $returnedstring" -ForegroundColor magenta  
write-host
```

```

1 function ApplyCase ([switch] $upper, [string] $str) {
2     if ( $upper ) {
3         $newstr = $str.ToUpper()
4     } else {
5         $newstr = $str.ToLower()
6     }
7     return $newstr
8 }
9
10
11 clear-host
12 write-host
13 $actualstring = 'THE QUICK BROWN FOX JUMPS OF THE LAZY DOG'
14
15 write-host "Function call with no switch parameter" -foreground yellow
16 $returnedstring = ApplyCase $actualstring
17 write-host "`tThe function ApplyCase: `n`t`t was called with $actualstring `n`t`t and returned"
18 write-host
19 write-host "Function call with -upper parameter" -foreground yellow
20 $actualstring = $returnedstring
21 $returnedstring = ApplyCase $actualstring -upper
22 write-host "`tThe function ApplyCase: `n`t`t was called with $actualstring `n`t`t and returned"
23 write-host
24

```

Function call with no switch parameter
The function ApplyCase:
was called with THE QUICK BROWN FOX JUMPS OF THE LAZY DOG
and returned the quick brown fox jumps of the lazy dog

Function call with -upper parameter
The function ApplyCase:
was called with the quick brown fox jumps of the lazy dog
and returned THE QUICK BROWN FOX JUMPS OF THE LAZY DOG

The next example is based upon the example in the previous section. In this example the function implements the switch parameter \$resolve. If \$resolve is set the function attempts to resolve the IP address to a name. If the host responds to a ping and is registered with DNS, the function returns the host name. If the function responds to a ping but is not registered, the function returns 'unknown'. If the host does not respond to a ping, the function returns 'dead'.

```

function ping-host (
    [switch] $resolve, [string] $ip = $($ throw "An IP address is
required" )
)

$IsValidIPAddress = $ip -match '\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}'
If (-not $IsValidIPAddress) {

```

```

        write-host "$ip is not a valid ip address" -ForegroundColor red
        return 'IP address is not valid'
    }

    if ($resolve) {
        $r = resolve-dnsname $ip -ErrorAction SilentlyContinue
        # check to see if we got a response
        if ($r) {
            $n = $r.NameHost
        } else {
            $n = 'Unknown'
        }
    } else {
        $n = 'No lookup done'
    }

    $r = test-connection $ip -count 1 -BufferSize 15 -ErrorAction
    SilentlyContinue
    if ($r.StatusCode -eq 0) {
        $msg = "$ip is alive and the name is $n"
    } else {
        $msg = "$ip is dead and the name is $n"
    }

    return $msg
}

# non-existent ip, do not resolve
$h = '172.16.1.2'
$r = ping-host $h
write-host "Host $h is $r" -ForegroundColor Yellow

# valid IP but not registered with DNS, try to resolve
$h = '207.75.134.1'
$r = ping-host $h -resolve
write-host "Host $h is $r" -ForegroundColor Yellow

# valid IP registered with DNS
$h = '198.111.176.6'
$r = ping-host $h -resolve
write-host "Host $h is $r" -ForegroundColor Yellow

```

Using Functions in a Pipeline

Tuesday, February 19, 2013
5:33 PM

The nature of a function is to take inputs usually from parameters, process the input, and return a value(s). For a function to be used in a pipeline, the function must be able to access the objects in the pipeline. This is accomplished using the `$input` variable. When used in a function the `$input` variable contains the objects coming down the pipeline and arriving at the function. When used in this fashion, the function is *blocking* which means all the objects must be collected first before the function executes.

```
function pinger {
    foreach ( $h in $input)  {
        test-connection $h -count 1 -erroraction silentlycontinue
    }
}
```

```
PS C:\Users>
PS C:\Users> function pinger {
>>>     foreach ( $h in $input)  {
>>>         test-connection $h -count 1 -erroraction silentlycontinue
>>>     }
PS C:\Users>
PS C:\Users> '8.8.8.8', '207.75.134.1','198.111.176.1' | pinger
Source          Destination      IPV4Address      IPV6Address
----          -----
IS017871        8.8.8.8          8.8.8.8
IS017871        207.75.134.1
IS017871        198.111.176.1
```

Once a function is written in the form, the only means available to pass parameters to the function is through the pipeline.

Filters

Tuesday, February 19, 2013
7:55 PM

The problem with using a function in a pipeline is that a function only runs once. This means the cmdlet preceding the function must accumulate all the objects and populate \$input before the function is executed. This may be problematic given the number of objects and size of each object. This also implies that a potentially significant delay occurs before output exits the pipeline. To remedy this, PowerShell implements **filters**.

The only syntactic difference between a function and a **filter** is the just the replacement of the keyword function with the keyword filter and using the \$input variable as a scalar that contains the current object in the pipeline. However the semantic differences are significant. A filter executes once for each object in the pipeline, i.e., **streaming**. This remedies the potential memory usage and output delay problems associated with functions.

Let's examine the example for the previous section. In this case the function is blocking. Notice that a foreach statement is required to iterate through all the elements in the \$input array.

```
function pinger {
    foreach ( $h in $input) {
        test-connection $h -count 1 -erroraction silentlycontinue
    }
}
```

Rewriting the above function as a *filter* eliminates the need for the foreach loop since, in this case, \$input contains the current object in the pipeline.

```
filter pinger {
    test-connection $input -count 1 -erroraction silentlycontinue
}
```

```

PS C:\Users>
PS C:\Users> filter pinger {
>>>     test-connection $input -count 1 -erroraction silentlycontinue
>>> }
PS C:\Users>
PS C:\Users> '8.8.8.8', '207.75.134.1', '198.111.176.1' | pinger

Source          Destination        IPV4Address        IPV6Address
-----          -----           8.8.8.8
IS017871       8.8.8.8
IS017871       207.75.134.1
IS017871       198.111.176.1

```

Let's look at another simple example of a function that works in a pipeline. This function resolves an IP address and displays the IP address and host name, if registered, on the console.

```

function resolver {
    foreach ( $h in $input)  {
        $r = resolve-dnsname $h -Erroraction SilentlyContinue
        if ($r) {
            $n = $r.NameHost
        } else {
            $n = 'Unknown'
        }
        write-host "The ip address is $h and the host name is $n" -
ForegroundColor Magenta
    }
}

```

```
1 function resolver {
2     foreach ( $h in $input) {
3         $r = resolve-dnsname $h -Erroraction SilentlyContinue
4         if ($r) {
5             $n = $r.NameHost
6         } else {
7             $n = 'Unknown'
8         }
9         write-host "The ip address is $h and the host name is $n" -Foregroundcolor Magenta
10    }
11 }
```

```
PS C:\Users>
PS C:\Users> '8.8.8.8', '198.111.176.6', '207.75.134.1', '198.111.176.7' | resolver
The ip address is 8.8.8.8 and the host name is google-public-dns-a.google.com
The ip address is 198.111.176.6 and the host name is www.wccnet.edu
The ip address is 207.75.134.1 and the host name is Unknown
The ip address is 198.111.176.7 and the host name is Unknown
```

In the above execution of the pipeline, all of the IP address are first accumulated into \$input then the resolver function is executed at the end of the pipeline. The above example uses on a few IP addresses so efficiency is not a consideration. But consider the case where hundreds of IP address are import from a csv file. In this case, a significant delay will occur before the first message is displayed.

Changing the function to a filter requires a more subtle but important change. Unlike the function, the filter is executed for every object in the pipeline. The changes required to the above function is to remove the foreach loop and use \$input directly. The variable \$input contains the current IP address in the pipeline.

```
filter resolver {
    $r = resolve-dnsname $input -Erroraction SilentlyContinue
    if ($r) {
        $n = $r.NameHost
    } else {
        $n = 'Unknown'
    }
    write-host "The ip address is $input and the host name is $n" -Foregroundcolor Magenta
}
```

```

1 filter resolver {
2     $r = resolve-dnsname $input -Erroraction SilentlyContinue
3     if ($r) {
4         $n = $r.NameHost
5     } else {
6         $n = 'Unknown'
7     }
8     write-host "The ip address is $input and the host name is $n" -ForegroundColor Magenta
9 }
10

```

```

PS C:\Users>
PS C:\Users> '8.8.8.8', '198.111.176.6', '207.75.134.1', '198.111.176.7' | resolve-dnsname
The ip address is 8.8.8.8 and the host name is google-public-dns-a.google.com
The ip address is 198.111.176.6 and the host name is www.wccnet.edu
The ip address is 207.75.134.1 and the host name is Unknown
The ip address is 198.111.176.7 and the host name is Unknown

```

Instead of using the \$input variable we may use the *this object variable* \$_ to represent the current object.

```

filter resolver {
    $r = resolve-dnsname $_ -Erroraction SilentlyContinue
    if ($r) {
        $n = $r.NameHost
    } else {
        $n = 'Unknown'
    }
    write-host "The ip address is $_ and the host name is $n" -ForegroundColor Magenta
}

```

```

PS C:\Users>
PS C:\Users> filter resolver {
>>>     $r = resolve-dnsname $_ -Erroraction SilentlyContinue
>>>     if ($r) {
>>>         $n = $r.NameHost
>>>     } else {
>>>         $n = 'Unknown'
>>>     }
>>>     write-host "The ip address is $_ and the host name is $n" -ForegroundColor Magenta
>>> }
PS C:\Users>
PS C:\Users> '8.8.8.8', '198.111.176.6', '207.75.134.1', '198.111.176.7' | resolve-dnsname
The ip address is 8.8.8.8 and the host name is google-public-dns-a.google.com
The ip address is 198.111.176.6 and the host name is www.wccnet.edu
The ip address is 207.75.134.1 and the host name is Unknown
The ip address is 198.111.176.7 and the host name is Unknown

```

Streaming Functions

Tuesday, February 19, 2013
8:23 PM

To mitigate the behavior of functions in a pipeline, PowerShell implements the begin, process, blocks. The begin script block executes once when the function is first executed. The process executes once for every object in the pipeline. The end script block executes once after all pipeline objects have been processed. Note, what is described in this section also applies to scripts. Scripts written with begin, process, and end scriptblock may be used in a pipeline.

```
function <function name> ( <parameter list> ) {  
    begin {  
        <scriptblock>  
    }  
  
    process {  
        <scriptblock>  
    }  
  
    end {  
        <scriptblock>  
    }  
}
```

This scriptblock is executed once regardless of the number of objects. Its intended use is to initialize function variables.

This scriptblock is executed once for each object in the pipeline

This scriptblock is executed once after all objects in the pipeline are processed

Let's modify the resolver function to work better by allowing streaming.

```
function resolver {  
    begin {  
        write-host "function initialization" -ForegroundColor Yellow  
        $nhost = 0  
    }  
    process {  
        $r = resolve-dnsname $_ -Erroraction SilentlyContinue  
        if ($r) {  
            $n = $r.NameHost  
        } else {  
            $n = 'Unknown'  
        }  
        write-host "The ip address is $_ and the host name is $n" -ForegroundColor Magenta  
        $nhost++  
    }  
    end {  
        write-host "The number of hosts processed is $nhost" -ForegroundColor yellow  
    }  
}
```

```
1 function resolver {  
2     begin {
```

```
3     write-host "function initialization" -ForegroundColor Yellow
4     $nhost = 0
5   }
6   process {
7     $r = resolve-dnsname $_ -Erroraction SilentlyContinue
8     if ($r) {
9       $n = $r.NameHost           # was an object returned
10    } else {
11      $n = 'Unknown'          # no
12    }
13    write-host "The ip address is $_ and the host name is $n" -ForegroundColor Magenta
14    $nhost++
15  }
16  end {
17    write-host "The number of hosts processed is $nhost" -ForegroundColor yellow
18  }
19 }
```

```
<
PS C:\Users>
PS C:\Users>
PS C:\Users> '8.8.8.8', '198.111.176.6', '207.75.134.1', '198.111.176.7' | resolver
function initialization
The ip address is 8.8.8.8 and the host name is google-public-dns-a.google.com
The ip address is 198.111.176.6 and the host name is www.wccnet.edu
The ip address is 207.75.134.1 and the host name is Unknown
The ip address is 198.111.176.7 and the host name is Unknown
The number of hosts processed is 4
```

Passing Parameters

Sunday, February 24, 2013
9:49 AM

Like functions scripts may receive parameters. Scripts may use formal parameters or may use the \$args array to access the actual parameters specified on the command line. If the script is receiving a variable number of parameters from the command line then formal parameters should not be used. The script must implement the \$args array.

Using the \$args array:

Given the following script saved as hello.ps1:

```
# was an parameter passed?
```

```
if ( $args ) # this is identical to $args -ne $null
    { write-host "Hello $args" -foreground magenta }      # yes
else
    { write-host "Hello world" -foreground yellow}        # no
```

This script may be run as follows:

```
1 # was an parameter passed?
2 if ( $args )
3     { write-host "Hello $args" -foreground magenta }      # yes
4 else
5     { write-host "Hello world" -foreground yellow}        # no
6
```

```
PS C:\users>
PS C:\users> .\hello.ps1
Hello world

PS C:\users> .\hello.ps1 Mike
Hello Mike
```

The next example scripts demonstrates multiple methods for working with the \$args array. The first example is not very useful but does provide an easy method for displaying the parameters passed to the script. Assuming the script below is named list-args.ps1:

```
write-host "The number of parameters passed is $($args.length)"
write-host "The parameters are $args" -foreground magenta
```

```
1 write-host "The number of parameters passed is $($args.length)" -foreground magenta
2 write-host "The parameters are $args" -foreground magenta
3

PS C:\users>
PS C:\users> .\list-args.ps1
The number of parameters passed is 0
The parameters are

PS C:\users> .\list-args.ps1 'par1'
The number of parameters passed is 1
The parameters are par1

PS C:\users> .\list-args.ps1 'par1' 5
The number of parameters passed is 2
The parameters are par1 5

PS C:\users> .\list-args.ps1 'par1' 5 'par3'
The number of parameters passed is 3
The parameters are par1 5 par3

PS C:\users> .\list-args.ps1 'par1' 5 'par3' 'multiple words but one string'
The number of parameters passed is 4
The parameters are par1 5 par3 multiple words but one string
```

The next example uses one write-host statement to display each parameter individually. If a fixed number of parameters are passed to a script, named parameters would be more appropriate. Assuming the script list-args.ps1 is modified as below.

```
write-host "The number of parameters passed is $($args.length)" -foreground magenta
write-host "args[0] is $($args[0])" -foreground magenta
write-host "args[1] is $($args[1])" -foreground magenta
write-host "args[2] is $($args[2])" -foreground magenta
write-host "args[3] is $($args[3])" -foreground magenta
```

Executing this version of the script produces the results below.

```

1 write-host "The number of parameters passed is $($args.length)" -foreground magenta
2 write-host "args[0] is $($args[0])" -foreground magenta
3 write-host "args[1] is $($args[1])" -foreground magenta
4 write-host "args[2] is $($args[2])" -foreground magenta
5 write-host "args[3] is $($args[3])" -foreground magenta
6

PS C:\Users>
PS C:\Users> .\list-args.ps1
The number of parameters passed is 0
args[0] is
args[1] is
args[2] is
args[3] is

PS C:\Users> .\list-args.ps1 'par1'
The number of parameters passed is 1
args[0] is par1
args[1] is
args[2] is
args[3] is

PS C:\Users> .\list-args.ps1 'par1' 5
The number of parameters passed is 2
args[0] is par1
args[1] is 5
args[2] is
args[3] is

PS C:\Users> .\list-args.ps1 'par1' 5 'par3'
The number of parameters passed is 3
args[0] is par1
args[1] is 5
args[2] is par3
args[3] is

PS C:\Users> .\list-args.ps1 'par1' 5 'par3' 'multiple words but one string'
The number of parameters passed is 4
args[0] is par1
args[1] is 5
args[2] is par3
args[3] is multiple words but one string

```

The next example illustrates using the foreach statement which is one of the preferred methods, the other is the for statement, for dealing with a variable number of parameters passed on the command line. Assuming the script list-args.ps1 contains the following.

```

write-host 'Using a foreach to list the parameters' -foreground magenta
write-host "The number of parameters passed is $($args.length)" -foreground magenta
foreach ( $p in $args ) {
    write-host "p is $p" -foreground magenta
}

```

```
}
```

Executing this script produces the following.

```
1 write-host 'Using a foreach to list the parameters' -foreground magenta
2 write-host "The number of parameters passed is $($args.length)" -foreground magenta
3 foreach ( $p in $args ) {
4     write-host "p is $p" -foreground magenta
5 }
6

PS C:\Users>
PS C:\Users> .\list-args.ps1
Using a foreach to list the parameters
The number of parameters passed is 0

PS C:\Users> .\list-args.ps1 'par1'
Using a foreach to list the parameters
The number of parameters passed is 1
p is par1

PS C:\Users> .\list-args.ps1 'par1' 5
Using a foreach to list the parameters
The number of parameters passed is 2
p is par1
p is 5

PS C:\Users> .\list-args.ps1 'par1' 5 'par3'
Using a foreach to list the parameters
The number of parameters passed is 3
p is par1
p is 5
p is par3

PS C:\Users> .\list-args.ps1 'par1' 5 'par3' 'multiple words but one string'
Using a foreach to list the parameters
The number of parameters passed is 4
p is par1
p is 5
p is par3
p is multiple words but one string
```

Finally this example use a counted loop (for loop) to display the parameters.

```
write-host 'Using a foreach to list the parameters' -foreground magenta
write-host "The number of parameters passed is $($args.length)" -foreground magenta
for ($i=0; $i -lt $args.length; $i++) {
    $p = $args[$i]
    write-host "p is $p" -foreground magenta
```

```
}
```

```
1 write-host 'Using a foreach to list the parameters' -foreground magenta
2 write-host "The number of parameters passed is $($args.length)" -foreground magenta
3 for ($i=0; $i -lt $args.length; $i++) {
4     $p = $args[$i]
5     write-host "p is $p" -foreground magenta
6 }
7

<
PS C:\Users>
PS C:\Users> .\list-args.ps1
Using a foreach to list the parameters
The number of parameters passed is 0

PS C:\Users> .\list-args.ps1 'par1'
Using a foreach to list the parameters
The number of parameters passed is 1
p is par1

PS C:\Users> .\list-args.ps1 'par1' 5
Using a foreach to list the parameters
The number of parameters passed is 2
p is par1
p is 5

PS C:\Users> .\list-args.ps1 'par1' 5 'par3'
Using a foreach to list the parameters
The number of parameters passed is 3
p is par1
p is 5
p is par3

PS C:\Users> .\list-args.ps1 'par1' 5 'par3' 'multiple words but one string'
Using a foreach to list the parameters
The number of parameters passed is 4
p is par1
p is 5
p is par3
p is multiple words but one string
```

One final example which is a rewrite of the elogs function found the Functions chapter. This script produces identical results as the function.

```
# script get-elog
#
# This script is run:   elogs <log name> <source1> [<source2>..<sourcen>]
#
```

```

# check to make sure parameters were passed
if ( -not $args ) {
    write-host "need to pass log name (s) and source(s)" -foreground red
    return
}

# get the log(s) name
$logs = $args[0]

# create an empty array
$sources = @()
# this loops thru the command line parameters starting with the second
for ( $i=1; $i -lt $args.length; $i++ ) {
    # append the command line parameter
    $sources += $args[$i]
}

# the -source parameter supports arrays
get-eventlog -logname $logs -source $sources | format-table -autosize

```

Using the named formal parameters:

Formal named parameters may be specified using the *param* statement. The above script may be re-written to use named parameters instead of the \$args array.

In the example below, we are using enhance syntax for the param statement. This is necessary so we may add properties to the parameters. The parameter \$name has a default value of "World". If not name is passed to the script, the \$name parameter is assigned the default value.

```

param (
    [Parameter()] $name = 'World'
)
$name = 'magenta'
if ($name -eq 'World') { $color = "green" }
write-host "Hello" $name -foreground $color

```

```
1 param (
2     [Parameter()] $name = 'World'
3 )
4 $color = 'magenta'
5 if ($name -eq 'World') { $color = "yellow" }
6 write-host "Hello" $name -foreground $color
7
```

```
PS C:\Users>
PS C:\Users> .\script.ps1
Hello World

PS C:\Users> .\script.ps1 Mike
Hello Mike
```

Parameters may be further qualified by setting properties. Many properties are available, the properties listed below are some of the more common parameter properties.

Mandatory: By default, all parameters are optional. When mandatory is specified, If the user omits the parameter, PowerShell prompts the user for the parameter.

In the example below, the parameter \$name is made mandatory. Notice that the default value is removed since the mandatory property and a default value are mutually exclusive.

```
param (
    [Parameter(Mandatory)] $name
)
$name = 'magenta'
if ($name -eq 'World') { $color = "green" }
write-host "Hello" $name -foreground $color
```

```

1 param (
2     [Parameter(Mandatory)] $name
3 )
4 $color = 'magenta'
5 if ($name -eq 'World') { $color = "green" }
6 write-host "Hello" $name -foreground $color
7

```

```

PS C:\Users>
PS C:\Users> .\script.ps1 Mike
Hello Mike

PS C:\Users> .\script.ps1
cmdlet script.ps1 at command pipeline position 1
Supply values for the following parameters:
name: Joe
Hello Joe

```

Ideally, it would nice to throw an error if the caller didn't supply an actual parameter. This is possible with the validation methods described later in this section.

Position: All parameters are both positional and named by default. The position property indicates the sequential order of the parameters. Once a parameter is specified with the position property, all remaining parameters become non-positional; which means either the position property is set for all of the parameters or the parameter must be named when the script is executed.

```

param (
    [parameter(Mandatory=$true, Position=0)] $p1,
    $p2='value2'
)
write-host "The value of parameter p1 is $p1" -ForegroundColor Magenta
write-host "The value of parameter p2 is $p2" -ForegroundColor Yellow

```

```

1 param (
2     [parameter(Mandatory=$true, Position=0)] $p1,
3     $p2='value2'
4 )
5 write-host "The value of parameter p1 is $p1" -ForegroundColor Magenta
6 write-host "The value of parameter p2 is $p2" -ForegroundColor Yellow
7

C:\>
PS C:\Users>
PS C:\Users> .\script.ps1 'The first parameter'
The value of parameter p1 is The first parameter
The value of parameter p2 is value2

PS C:\Users> .\script.ps1 'The first parameter' 'The second parameter'
C:\Users\script.ps1 : A positional parameter cannot be found that accepts argument 'The second parameter'.
At line:1 char:1
+ .\script.ps1 'The first parameter' 'The second parameter'
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo          : InvalidArgument: (:) [script.ps1], ParameterBindingException
+ FullyQualifiedErrorId : PositionalParameterNotFound,script.ps1

PS C:\Users> .\script.ps1 'The first parameter' -p2 'The second parameter'
The value of parameter p1 is The first parameter
The value of parameter p2 is The second parameter

PS C:\Users> .\script.ps1 -p2 'The second parameter'
cmdlet script.ps1 at command pipeline position 1
Supply values for the following parameters:
p1: The mandatory parameter
The value of parameter p1 is The mandatory parameter
The value of parameter p2 is The second parameter

PS C:\Users> .\script.ps1 -p1 'The first parameter' -p2 'The second parameter'
The value of parameter p1 is The first parameter
The value of parameter p2 is The second parameter

```

The problem occurs when more than two parameters are used. In this case, all parameters require a Position property or the script must be executed with named parameters.

```

param (
    [parameter(Mandatory=$true, Position=0)] $p1,
    $p2='value2',
    $p3 ='value3'
)
write-host "The value of parameter p1 is $p1" -ForegroundColor Magenta
write-host "The value of parameter p2 is $p2" -ForegroundColor Yellow
write-host "The value of parameter p3 is $p3" -ForegroundColor Yellow

```

The possible ways this script may be executed are:

```

1 param (
2     [parameter(Mandatory=$true, Position=0)] $p1,
3     $p2='value2',
4     $p3 ='value3'
5 )
6 write-host "The value of parameter p1 is $p1" -ForegroundColor Magenta
7 write-host "The value of parameter p2 is $p2" -ForegroundColor Yellow
8 write-host "The value of parameter p3 is $p3" -ForegroundColor Cyan
9
10
PS C:\Users>
PS C:\Users> .\script.ps1 -p1 'The first parameter' -p2 'The second parameter' -p3 'The third parameter'
The value of parameter p1 is The first parameter
The value of parameter p2 is The second parameter
The value of parameter p3 is The third parameter

PS C:\Users> .\script.ps1 -p2 'The second parameter' -p3 'The third parameter'
cmdlet script.ps1 at command pipeline position 1
Supply values for the following parameters:
p1: The mandatory parameter
The value of parameter p1 is The mandatory parameter
The value of parameter p2 is The second parameter
The value of parameter p3 is The third parameter

```

This behavior may be corrected by specifying positions for all parameters:

```

param (
    [parameter(Mandatory=$true, Position=0)] $p1,
    [parameter(Position=1)]$p2='value2',
    [parameter(Position=2)]$p3='value3'
)
write-host "The value of parameter p1 is $p1" -ForegroundColor Magenta
write-host "The value of parameter p2 is $p2" -ForegroundColor Yellow
write-host "The value of parameter p3 is $p3" -ForegroundColor Cyan

```

The possible ways this script may now be executed are:

```

1 param (
2     [parameter(Mandatory=$true, Position=0)] $p1,
3     [parameter(Position=1)] $p2='value2',
4     [parameter(Position=2)] $p3='value3'
5   )
6 write-host "The value of parameter p1 is $p1" -ForegroundColor Magenta
7 write-host "The value of parameter p2 is $p2" -ForegroundColor Yellow
8 write-host "The value of parameter p3 is $p3" -ForegroundColor Cyan
9
10
PS C:\Users>
PS C:\Users> .\script.ps1 'The first parameter' 'The second parameter' 'The third parameter'
The value of parameter p1 is The first parameter
The value of parameter p2 is The second parameter
The value of parameter p3 is The third parameter

PS C:\Users> .\script.ps1 -p2 'The second parameter' -p3 'The third parameter'
cmdlet script.ps1 at command pipeline position 1
Supply values for the following parameters:
p1: The mandatory parameter
The value of parameter p1 is The mandatory parameter
The value of parameter p2 is The second parameter
The value of parameter p3 is The third parameter

```

ValuefromPipeline: This property make the parameter settable from the pipeline.

The example below is an implementation of the resolver function as a script.

```

param ( $ipaddr )
begin {
    write-host "script initialization" -ForegroundColor Yellow
    $nhost = 0
}
process {
    $r = resolve-dnsname $ipaddr -Erroraction SilentlyContinue
    if ($r) {
        $n = $r.NameHost
    } else {
        $n = 'Unknown'
    }
    write-host "The ip address is $ipaddr and the host name is $n" -
    ForegroundColor Magenta
    $nhost++ # same as $nhost = $nhost + 1
}
end {
    write-host "The number of hosts processed is $nhost" -
    ForegroundColor yellow
}

```

Executing the script from the command line produces the desired results.

```
1 param ( $ipaddr )
2 begin {
3     write-host "script initialization" -ForegroundColor Yellow
4     $nhost = 0
5 }
6 process {
7     $r = resolve-dnsname $ipaddr -Erroraction SilentlyContinue
8     if ($r) {
9         $n = $r.NameHost
10    } else {
11        $n = 'Unknown'
12    }
13    write-host "The ip address is $ipaddr and the host name is $n" -ForegroundColor Magenta
14    $nhost++ # same as $nhost = $nhost + 1
15 }
16 end {
17     write-host "The number of hosts processed is $nhost" -ForegroundColor yellow
18 }
19
```

```
PS C:\Users>
PS C:\Users> .\resolver.ps1 198.111.176.6
script initialization
The ip address is 198.111.176.6 and the host name is www.wccnet.org
The number of hosts processed is 1
```

Executing the script in a pipeline as below doesn't work.

```
PS C:\Users>
PS C:\Users> '8.8.8.8', '207.75.134.1' , '198.111.176.6' , '198.111.176.7' , '198.111.176.8' | .\resolver.ps1
script initialization
Resolve-DnsName : Cannot validate argument on parameter 'Name'. The argument is null or empty. Provide an argument that is not null or empty, and then try the command again.
At C:\Users\resolver.ps1:7 char:26
+     $r = resolve-dnsname $ipaddr -Erroraction SilentlyContinue
+               ~~~~~
+ CategoryInfo          : InvalidData: (:) [Resolve-DnsName], ParameterBindingValidationException
+ FullyQualifiedErrorId : ParameterArgumentValidationError,Microsoft.DnsClient.Commands.ResolveDnsName

The ip address is  and the host name is Unknown
Resolve-DnsName : Cannot validate argument on parameter 'Name'. The argument is null or empty. Provide an argument that is not null or empty, and then try the command again.
At C:\Users\resolver.ps1:7 char:26
+     $r = resolve-dnsname $ipaddr -Erroraction SilentlyContinue
+               ~~~~~
+ CategoryInfo          : InvalidData: (:) [Resolve-DnsName], ParameterBindingValidationException
+ FullyQualifiedErrorId : ParameterArgumentValidationError,Microsoft.DnsClient.Commands.ResolveDnsName

The ip address is  and the host name is Unknown
Resolve-DnsName : Cannot validate argument on parameter 'Name'. The argument is null or empty. Provide an argument that is not null or empty, and then try the command again.
At C:\Users\resolver.ps1:7 char:26
+     $r = resolve-dnsname $ipaddr -Erroraction SilentlyContinue
+               ~~~~~
+ CategoryInfo          : InvalidData: (:) [Resolve-DnsName], ParameterBindingValidationException
+ FullyQualifiedErrorId : ParameterArgumentValidationError,Microsoft.DnsClient.Commands.ResolveDnsName
```

We can correct this behavior by setting the *ValueFromPipeline* property for the parameter.

```
param (
    [parameter (ValueFromPipeLine=$true,Mandatory=$true)] $ipaddr
)
begin {
    write-host "script initialization" -ForegroundColor Yellow
    $nhost = 0
}
process {
    $r = resolve-dnsname $ipaddr -Erroraction SilentlyContinue
    if ($r) {
        $n = $r.NameHost
    } else {
        $n = 'Unknown'
    }
    write-host "The ip address is $ipaddr and the host name is $n" -
ForegroundColor Magenta
    $nhost++ # same as $nhost = $nhost + 1
}
end {
    write-host "The number of hosts processed is $nhost" -
ForegroundColor yellow
}
```

Executing the script produces the correct result.

```

1 param (
2     [parameter (ValueFromPipeline=$true,Mandatory=$true)] $ipaddr
3 )
4 begin {
5     write-host "script initialization" -ForegroundColor Yellow
6     $nhost = 0
7 }
8 process {
9     $r = resolve-dnsname $ipaddr -Erroraction SilentlyContinue
10    if ($r) {
11        $n = $r.NameHost
12    } else {
13        $n = 'Unknown'
14    }
15    write-host "The ip address is $ipaddr and the host name is $n" -ForegroundColor Magenta
16    $nhost++ # same as $nhost = $nhost + 1
17 }
18 end {
19     write-host "The number of hosts processed is $nhost" -ForegroundColor yellow
20 }
21

```

```

PS C:\Users>
PS C:\Users> '8.8.8.8', '207.75.134.1' , '198.111.176.6', '198.111.176.7', '198.111.176.8' | .\resolver.ps1
script initialization
The ip address is 8.8.8.8 and the host name is google-public-dns-a.google.com
The ip address is 207.75.134.1 and the host name is Unknown
The ip address is 198.111.176.6 and the host name is www.wccnet.org
The ip address is 198.111.176.7 and the host name is cidermill.wccnet.edu
The ip address is 198.111.176.8 and the host name is westland.wccnet.edu
The number of hosts processed is 5

```

Parameter Validation Methods:

PowerShell provides a number of parameter validation methods. This simplifies the script development process by reducing the number of validation steps that the script writer has to develop. The following are some of the validation attributes that are available.

PowerShell treats empty, null, and unspecified parameters differently. As can be seen later "unspecified" is neither empty nor null.

ValidateNotNull: This method validates that the parameters is not null.

```

param (
    [ValidateNotNull()] $p1
)
write-host "The value of the parameter is $p1 "

```

```
1 param (
2     [ValidateNotNull()]] $p1
3 )
4 write-host "The value of the parameter is $p1"
5
6

PS C:\Users>
PS C:\Users> .\script.ps1
The value of the parameter is

PS C:\Users> $n = $null

PS C:\Users> .\script.ps1 $n
C:\Users\script.ps1 : Cannot validate argument on parameter 'p1'. The argument is null. Provide a valid value
argument, and then try running the command again.
At line:1 char:14
+ .\script.ps1 $n
+ ~~~~~
+ CategoryInfo          : InvalidData: (:) [script.ps1], ParameterBindingValidationException
+ FullyQualifiedErrorId : ParameterArgumentValidationError,script.ps1
```

The first execution of the script does not result in an error since the parameter is unspecified. In the second execution, a null value is passed to the script triggering the validation error.

ValidateNotNullorEmpty: This attribute prevents null or empty parameters.

```
param (
    [ValidateNotNullorEmpty()] $p1

)
write-host "The value of the parameter is $p1 "
```

```
1 param (
2     [ValidateNotNullOrEmpty()] $p1
3
4 )
5 write-host "The value of the parameter is $p1 "
6

PS C:\Users>
PS C:\Users> .\script.ps1
The value of the parameter is

PS C:\Users> .\script.ps1 ''
C:\Users\script.ps1 : Cannot validate argument on parameter 'p1'. The argument is null or empty. Provide a not null or empty, and then try the command again.
At line:1 char:14
+ .\script.ps1 ''
+
+ CategoryInfo          : InvalidData: (:) [script.ps1], ParameterBindingValidationException
+ FullyQualifiedErrorId : ParameterArgumentValidationError,script.ps1

PS C:\Users> .\script.ps1 $n
C:\Users\script.ps1 : Cannot validate argument on parameter 'p1'. The argument is null or empty. Provide a not null or empty, and then try the command again.
At line:1 char:14
+ .\script.ps1 $n
+
+ CategoryInfo          : InvalidData: (:) [script.ps1], ParameterBindingValidationException
+ FullyQualifiedErrorId : ParameterArgumentValidationError,script.ps1
```

Handling Errors

Sunday, March 31, 2013
1:33 PM

Powershell provides the \$ErrorActionPreference variable and the -ErrorAction switch to control the way errors are handled.

\$ErrorActionPreference indicates to PS how to proceed when an error occurs. The default setting is "Continue" which means that the execution script will continue to run regardless of the errors that may occur. While this may be considered dangerous, consider the case when a copy is in progress, should the inability to copy one file cause the script to abort?

\$ErrorActionPreference:

In the example below the files do not exist. With "Continue" as the default setting for \$ErrorActionPreference, both cmdlets are executed.

```
remove-item x.txt
remove-item y.txt
```

```
1
2 set-location c:\users
3
4 remove-item x.txt
5 remove-item y.txt
6 |>
<
PS C:\users>
PS C:\users>
set-location c:\users

remove-item x.txt
remove-item y.txt

remove-item : Cannot find path 'C:\users\x.txt' because it does not exist.
At line:4 char:1
+ remove-item x.txt
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (C:\users\x.txt:String) [Remove-Item], ItemN
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.RemoveItemCommand

remove-item : Cannot find path 'C:\users\y.txt' because it does not exist.
At line:5 char:1
+ remove-item y.txt
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (C:\users\y.txt:String) [Remove-Item], ItemN
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.RemoveItemCommand
```

Changing \$ErrorActionPreference to "Stop" causes the script to stop after the first error occurs.

```
remove-item x.txt
remove-item y.txt
```

```
1
2 set-location c:\users
3
4 $ErrorActionPreference = 'stop'
5
6 remove-item x.txt
7 remove-item y.txt
8
```

```
<
PS C:\users>
PS C:\users>
set-location c:\users

$ErrorActionPreference = 'stop'

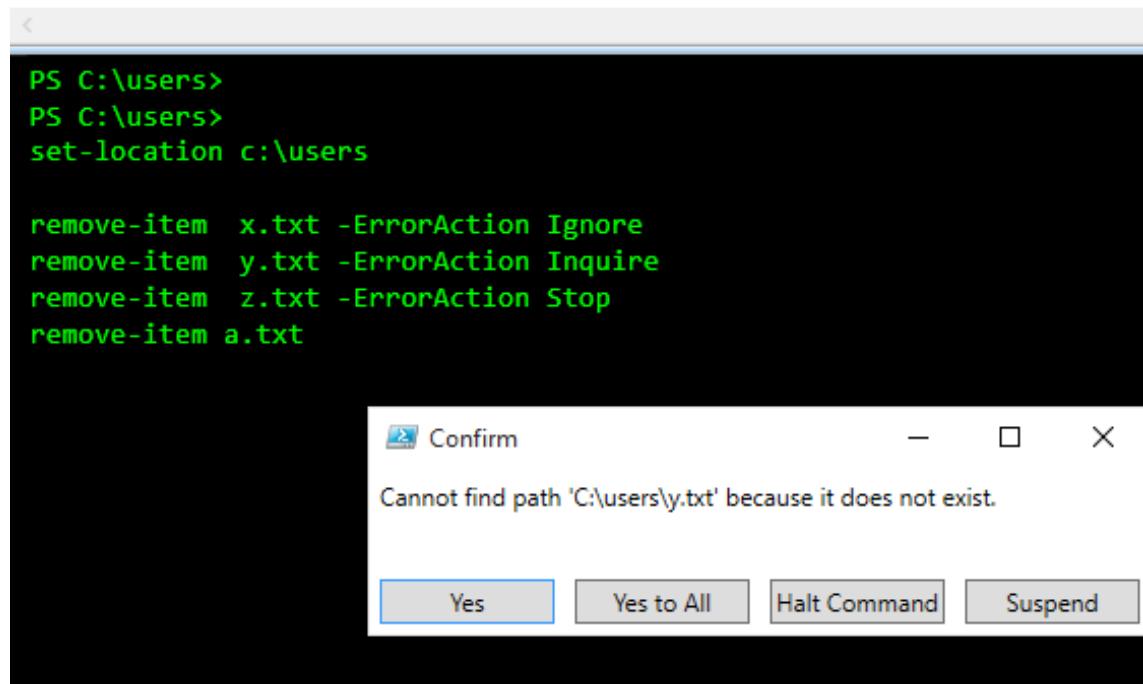
remove-item x.txt
remove-item y.txt

remove-item : Cannot find path 'C:\users\x.txt' because it does not exist.
At line:6 char:1
+ remove-item x.txt
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (C:\users\x.txt:String) [Remove-Item], It
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.RemoveItemComm
```

\$ErrorActionPreference applies globally to all commands in the script. The error action for individual cmdlets may be controlled with the -ErrorAction parameter. The error is ignored in the first remove-item cmdlet execution. The second execution would result in a confirmation message box as shown below. The third would throw an error and stop the script. The fourth would not be executed.

```
remove-item x.txt -ErrorAction Ignore
remove-item y.txt -ErrorAction Inquire
remove-item z.txt -ErrorAction Stop
remove-item a.txt
```

```
1
2 set-location c:\users
3
4 remove-item x.txt -ErrorAction Ignore
5 remove-item y.txt -ErrorAction Inquire
6 remove-item z.txt -ErrorAction Stop
7 remove-item a.txt
8
```



The screenshot shows a PowerShell window with the following command history:

```
PS C:\users>
PS C:\users>
set-location c:\users

remove-item x.txt -ErrorAction Ignore
remove-item y.txt -ErrorAction Inquire
remove-item z.txt -ErrorAction Stop
remove-item a.txt
```

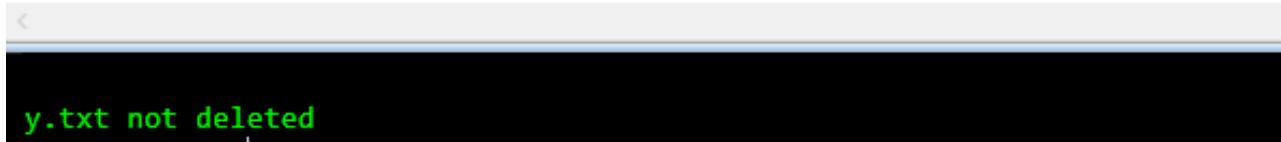
A 'Confirm' dialog box is displayed in the foreground, containing the message: "Cannot find path 'C:\users\y.txt' because it does not exist." The dialog has four buttons: "Yes" (highlighted in blue), "Yes to All", "Halt Command", and "Suspend".

\$? variable:

PowerShell also provides the error status `$?` so the disposition of the last executed cmdlet may be interrogated. If the cmdlet executes successfully, `$?` is set to `$true` otherwise it is set to `false`.

```
set-location c:\users
clear-host
write-host
remove-item y.txt -ErrorAction Ignore
if ( $? -eq $true ) { write-host "y.txt deleted" }
else { write-host "y.txt not deleted" }
```

```
1
2 set-location c:\users
3 clear-host
4 write-host
5 remove-item y.txt -ErrorAction Ignore
6 if ( $? -eq $true ) { write-host "y.txt deleted"}
7 else { write-host "y.txt not deleted" }
8
```



```
y.txt not deleted
```

ErrorVariable:

The ErrorVariable switch stores error information in a user-specified variable in the event of an error. This user-specified variable is an array to which error information is appended whenever the variable is used in a cmdlet.

```
clear-host
write-host
write-host 'Attempting to remove y.txt' -ForegroundColor yellow
remove-item y.txt -ErrorAction SilentlyContinue -ErrorVariable err
if ( $err[0] ) {
    $t = $err[0].TargetObject
    $e = $err[0].Exception.Message
    write-host "`tError occurred: `n`t`t targetobject : $t `n`t`t
exception $e" -ForegroundColor Red
}

write-host 'Attempting to remove z.txt' -ForegroundColor Yellow
remove-item z.txt -ErrorAction SilentlyContinue -ErrorVariable +err
# append to the error array
if ( $err[1] ) {
    $t = $err[1].TargetObject
    $e = $err[1].Exception.Message
    write-host "`tError occurred: `n`t`t targetobject : $t `n`t`t
exception $e" -ForegroundColor Red
}
```

```

1
2 clear-host
3 write-host
4 write-host 'Attempting to remove y.txt' -ForegroundColor yellow
5 remove-item y.txt -ErrorAction SilentlyContinue -ErrorVariable err
6 if ( $err[0] ) {
7     $t = $err[0].TargetObject
8     $e = $err[0].Exception.Message
9     write-host "`tError occurred: `n`t`t targetobject : $t `n`t`t exception $e" -ForegroundColor red
10 }
11
12 write-host 'Attempting to remove z.txt' -ForegroundColor Yellow
13 remove-item z.txt -ErrorAction SilentlyContinue -ErrorVariable +err # append to the error array
14 if ( $err[1] ) {
15     $t = $err[1].TargetObject
16     $e = $err[1].Exception.Message
17     write-host "`tError occurred: `n`t`t targetobject : $t `n`t`t exception $e" -ForegroundColor red
18 }
19

```

```

Attempting to remove y.txt
Error occurred:
    targetobject : C:\users\y.txt
    exception Cannot find path 'C:\users\y.txt' because it does not exist.

Attempting to remove z.txt
Error occurred:
    targetobject : C:\users\z.txt
    exception Cannot find path 'C:\users\z.txt' because it does not exist.

```

Trap statement:

The trap statement is used to execute a scriptblock when an error occurs. In the function below, the trap script block is executed whenever an error occurs in the function. The only possibility for an error is on the remove-item statement. Since the default \$ErrorActionPreference is "Continue", it's imperative that -ErrorAction Stop be specified for the remove-item cmdlet. Anything other than "Stop" will prevent the trap from executing.

```

# Script: delete_file <file name>
param (
    [Parameter(Mandatory)] $filename
)

trap {
    write-host "$filename not found" -foregroundcolor Magenta
}

```

```

    return
}

write-host "Attempting to remove $filename"
remove-item $filename -ErrorAction Stop
write-host "File $filename removed" -ForegroundColor yellow

```

```

1 # Script: delete_file <file name>
2 param (
3     [Parameter(Mandatory)] $filename
4 )
5
6 trap {
7     write-host "$filename not found" -foregroundcolor Magenta
8     return
9 }
10
11 write-host "Attempting to remove $filename"
12 remove-item $filename -ErrorAction Stop
13 write-host "File $filename removed" -ForegroundColor yellow

```

```

PS C:\Users>
PS C:\Users> .\delete_file.ps1 .\bar.txt
Attempting to remove .\bar.txt
File .\bar.txt removed

PS C:\Users> .\delete_file.ps1 .\bar.txt
Attempting to remove .\bar.txt
.\bar.txt not found

```

A trap statement is active for the scope in which it is defined. In the example below, the trap is defined within the scope of the function and hence does not trap the error in the main block.

```

# Script: delete_file <file name>
param (
    [Parameter(Mandatory)] $filename
)

function foo {

    trap {
        write-host "Trapped error in function"
    }
    write-host "In function"
}

```

```

foo    # call function foo
write-host "Attempting to remove $filename"
remove-item $filename -ErrorAction Stop
write-host "You should not see this message"

```

```

1 # Script: delete_file <file name>
2 param (
3     [Parameter(Mandatory)] $filename
4 )
5
6 function foo {
7
8     trap {
9         write-host "Trapped error in function"
10    }
11    write-host "In function"
12 }
13
14 foo    # call function foo
15 write-host "Attempting to remove $filename"
16 remove-item $filename -ErrorAction Stop
17 write-host "You should not see this message"
18

```

```

PS C:\Users>
PS C:\Users> .\delete_file.ps1 .\bar.txt
In function
Attempting to remove .\bar.txt
remove-item : Cannot find path 'C:\Users\bar.txt' because it does not exist.
At C:\Users\delete_file.ps1:16 char:1
+ remove-item $filename -ErrorAction Stop
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (C:\Users\bar.txt:String) [Remove-Item]
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.RemoveItem

```

In contrast to the above example, the example below demonstrates the a global trap. Since the trap statement is defined in the outer block, it applies to all scopes within the script. So the error is trapped in the main block. Since the trap ends with a return, control is returned to the calling environment which essentially exits the script.

```

# Script: delete_file <file name>
param (
    [Parameter(Mandatory)] $filename
)

```

```

function foo {
    write-host "In function"
}

trap {
    write-host "Trapped error in the main block" -ForegroundColor Magenta
    return
}

foo # call function foo
write-host "Attempting to remove $filename"
remove-item $filename -ErrorAction stop
write-host "You should not see this message"

```

```

1 # Script: delete_file <file name>
2 param (
3     [Parameter(Mandatory)] $filename
4 )
5
6 function foo {
7     write-host "In function"
8 }
9
10 trap {
11     write-host "Trapped error in the main block" -ForegroundColor Magenta
12     return
13 }
14
15 foo # call function foo
16 write-host "Attempting to remove $filename"
17 remove-item $filename -ErrorAction stop
18 write-host "You should not see this message"

```

```

PS C:\Users>
PS C:\Users> .\delete_file.ps1 .\bar.txt
In function
Attempting to remove .\bar.txt
Trapped error in the main block

```

Since the trap is global if an error occurs in the function, the trap in the main block traps it. Because the trap ends with a continue, control is returned to the next statement after the function call.

```
# Script: delete_file <file name>
```

```

param (
    [Parameter(Mandatory)] $filename
)

function foo {
    write-host "In function"
    write-host "Attempting to remove $filename"
    remove-item $filename -ErrorAction stop
    write-host "You should not see this message"
}

trap {
    write-host "Trapped error in the main block" -ForegroundColor Magenta
    continue
}

foo # call function foo
write-host "You will see this message because the trap has continue"

```

```

1 # Script: delete_file <file name>
2 param (
3     [Parameter(Mandatory)] $filename
4 )
5
6 function foo {
7     write-host "In function"
8     write-host "Attempting to remove $filename"
9     remove-item $filename -ErrorAction stop
10    write-host "You should not see this message"
11 }
12
13 trap {
14     write-host "Trapped error in the main block" -ForegroundColor Magenta
15     continue
16 }
17
18 foo # call function foo
19 write-host "You will see this message because the trap has continue"

```

```

PS C:\Users>
PS C:\Users> .\delete_file.ps1 .\bar.txt
In function
Attempting to remove .\bar.txt
Trapped error in the main block
You will see this message because the trap has continue

```

Error information is available in the trap script block using the *this* object `$_.`

```
# Script: delete_file <file name>
param (
    [Parameter(Mandatory)] $filename
)

function foo {
    write-host "In function"
    write-host "Attempting to remove $filename"
    remove-item $filename -ErrorAction stop
    write-host "You should not see this message"
}

trap {
    write-host "Trapped error in the main block" -ForegroundColor Magenta
    write-host "Error Category: $($_.CategoryInfo.Category)" -ForegroundColor Magenta
    write-host "Error Type: $($_.Exception.GetType().FullName)" -ForegroundColor Magenta
    write-host "Id: $($_.FullyQualifiedErrorID)" -ForegroundColor Magenta
    write-host "'Message: $($_.Exception.Message)" -ForegroundColor Magenta
    continue
}

foo # call function foo
write-host "You will see this message because the trap has continue"
```

```

6 function foo {
7     write-host "In function"
8     write-host "Attempting to remove $filename"
9     remove-item $filename -ErrorAction stop
10    write-host "You should not see this message"
11 }
12
13 trap {
14     write-host "Trapped error in the main block" -ForegroundColor Magenta
15     write-host "Error Category: $($_.CategoryInfo.Category)" -ForegroundColor Magenta
16     write-host "Error Type: $($_.Exception.GetType().FullName)" -ForegroundColor Magenta
17     write-host "Id: $($_.FullyQualifiedErrorID)" -ForegroundColor Magenta
18     write-host "'Message: $($_.Exception.Message)" -ForegroundColor Magenta
19     continue
20 }
21
22 foo # call function foo
23 write-host "You will see this message because the trap has continue"

```

```

PS C:\Users>
PS C:\Users> .\delete_file.ps1 bar.txt
In function
Attempting to remove bar.txt
Trapped error in the main block
Error Category: ObjectNotFound
Error Type: System.Management.Automation.ItemNotFoundException
Id: PathNotFound,Microsoft.PowerShell.Commands.RemoveItemCommand
'Message: Cannot find path 'C:\Users\bar.txt' because it does not exist.
You will see this message because the trap has continue

```

Try/Catch:

The try/catch statement allows errors to be trapped for specific statements. In the example below, the code in the try script block executes if no error occurs. If any error occurs in the try script block, the catch script block executes.

```

# Script: delete_file <file name>
param (
    [Parameter(Mandatory)] $filename
)

write-host "Attempting to remove $filename"
try {

```

```

remove-item $filename -ErrorAction stop
write-host "File $filename removed" -ForegroundColor yellow
}
catch {
    write-host 'Caught error' -ForegroundColor Magenta
    write-host "Message: $($_.Exception.Message)" -ForegroundColor Magenta
}

```

In the execution illustration below, foo.txt exists so the all cmdlets in the try block execute. In the second execution, bar.txt does not exists so the attempt by remove-item to delete the file causes the catch script block to execute.

```

1 # Script: delete_file <file name>
2 param (
3     [Parameter(Mandatory)] $filename
4 )
5
6 write-host "Attempting to remove $filename"
7 try {
8     remove-item $filename -ErrorAction stop
9     write-host "File $filename removed" -ForegroundColor yellow
10 }
11 catch {
12     write-host 'Caught error' -ForegroundColor Magenta
13     write-host "Message: $($_.Exception.Message)" -ForegroundColor Magenta
14 }

```

```

PS C:\Users>
PS C:\Users> .\delete_file.ps1 foo.txt
Attempting to remove foo.txt
File foo.txt removed

PS C:\Users> .\delete_file.ps1 bar.txt
Attempting to remove bar.txt
Caught error
Message: Cannot find path 'C:\Users\bar.txt' because it does not exist.

```

Throw:

The throw causes an error to occur. Throw will display the specified message. A .Net exception type may be specified which provides additional information as to the nature of the error.

In the script below, the \$filename parameter is required. Instead of using the Mandatory property, we use the throw to throw an error if the file name is not specified on the command line. In the script, we test for the existence of the file. If the file does not exist, the script throws a .Net exception.

```
# Script: delete_file <file name>
param (
    [Parameter()] $filename = $(throw "a file name is
required")
)

write-host "Attempting to remove $filename"

if (test-path $filename) {
    try {
        remove-item $filename -ErrorAction stop
        write-host "File $filename removed" -ForegroundColor yellow
    }
    catch {
        write-host 'Caught error' -ForegroundColor Magenta
        write-host "Message: $($_.Exception.Message)" -ForegroundColor
Magenta
    }
} else {
    throw [System.IO.FileNotFoundException] "File $filename does not
exist"
}
```

In the first execution of the script below, the file name is not specified on the command line so the parameter check throws the error. The second execution example shows what happens when the file to be deleted exists. The last execution example show the .Net error thrown because the file does not exist.

```
1 # Script: delete_file <file name>
2 param (
3     [Parameter()] $filename = $(throw "a file name is required")
4 )
5
6 write-host "Attempting to remove $filename"
7
8 if (test-path $filename) {
9     try {
10         remove-item $filename -ErrorAction stop
11         write-host "File $filename removed" -ForegroundColor yellow
12     }
13     catch {
14         write-host 'Caught error' -ForegroundColor Magenta
15         write-host "Message: $($_.Exception.Message)" -ForegroundColor Magenta
16     }
17 } else {
18     throw [System.IO.FileNotFoundException] "File $filename does not exist"
19 }
```

```
PS C:\Users>
PS C:\Users> .\delete_file.ps1    # no file name specified, scripts throws error
a file name is required
At C:\Users\delete_file.ps1:3 char:40
+ ...             [Parameter()] $filename = $(throw "a file name is required")
+ ~~~~~~
+ CategoryInfo          : OperationStopped: (a file name is required:String) [], Run
+ FullyQualifiedErrorId : a file name is required

PS C:\Users> .\delete_file.ps1  .\foo.txt  # file exists
Attempting to remove .\foo.txt
File .\foo.txt removed

PS C:\Users> .\delete_file.ps1  .\bar.txt  # file does not exist
Attempting to remove .\bar.txt
File .\bar.txt does not exist
At C:\Users\delete_file.ps1:18 char:2
+ throw [System.IO.FileNotFoundException] "File $filename does not ex ...
+ ~~~~~~
+ CategoryInfo          : OperationStopped: (:) [], FileNotFoundException
+ FullyQualifiedErrorId : File .\bar.txt does not exist
```

Using [cmdletbinding]

Sunday, March 31, 2013
10:58 AM

All cmdlets have a number of advanced features including parameter checking, whatif, confirm, and other parameters that provide advanced functionality.

Under normal circumstances, the script developer would have to write code implement these features in their scripts. With [cmdletbinding] these features are made available to the developer without requiring additional code in the scripts.

Whatif is one feature that should always be implemented in scripts. Whatif allows the effect of the cmdlet to be observed without actually performing the cmdlet action.

The following script creates a drive mapping to a directory and uses a traditional approach for implementing whatif and confirm:

```
param (
    [parameter(Mandatory=$true, Position=0)] [string] $drivename,
    [parameter(Mandatory=$true, Position=1)] [string] $target,
    [parameter(Mandatory=$false, Position=2)] [switch] $whatif,
    [parameter(Mandatory=$false, Position=3)] [switch] $confirm
)

If ($whatif)  {
    write-host "WhatIf: create network drive $drivename at
destination $target"
} else {
    if ($confirm)  {
        $ans = read-host "Confirm: create network drive $drivename at
destination $target"
        if ($ans -eq 'Y' -or $ans -eq 'Yes') {
            New-PSDrive $drivename FileSystem $target
        } else {
            write-host "Drive mapping not done"
        }
    } else {
        New-PSDrive $drivename FileSystem $target
    }
}
```

```
    }  
}
```

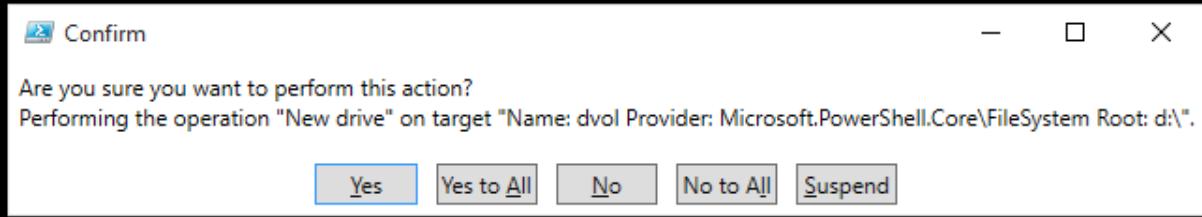
Using [cmdletbinding(SupportsShouldProcess=\$true)] the above script may be reduced to the code below since whatif and confirm are automatically provided.

```
[cmdletbinding(SupportsShouldProcess=$True)]  
param (  
    [parameter(Mandatory=$true, Position=0)] [string] $drivename,  
    [parameter(Mandatory=$true, Position=1)] [string] $target  
)  
  
New-PSDrive $drivename FileSystem $target
```

```
1 [cmdletbinding(SupportsShouldProcess=$True)]  
2 param (  
3     [parameter(Mandatory=$true, Position=0)] [string] $drivename,  
4     [parameter(Mandatory=$true, Position=1)] [string] $target  
5 )  
6  
7 New-PSDrive $drivename FileSystem $target
```



```
PS C:\Users>  
PS C:\Users> .\new-drive.ps1 dvol 'd:\' -whatif  
What if: Performing the operation "New drive" on target "Name: dvol Provider: Microsoft.PowerShell.Core\FileSystem Root: d\".  
  
PS C:\Users> .\new-drive.ps1 dvol 'd:\' -confirm
```

Another advantage of [cmdletbinding] is "SupportsPaging" which provides access to the parameters:

First: Intended to indicate to the script to process only the first n objects.

Skip: Intended to indicate to the script to skip the first n objects.

IncludeTotalCount: Informs the script to maintain an object count at report it

These switches provide information to the script. As seen in the next example, the script developer must develop the code to use the parameters. This script illustrates the uses of -First and -IncludeTotalCount.

```
[CmdletBinding(SupportsPaging = $true)]
param (
    [parameter (ValueFromPipeLine=$true, Mandatory=$true)] $num
)
begin { # initialize, happens once at the beginning
    $sum = $nnum = $rawcnt = 0
}

process { # happens for each object in the pipeline
    $rawcnt++

    if ($rawcnt -gt $PSCmdlet.PagingParameters.Skip) {
        $sum += $num
        $nnum++
    }
}

end { # happens once at the end
    write-host "the sum is $sum"
    if ($PSCmdlet.PagingParameters.IncludeTotalCount) {
        write-host "$nnum numbers processed"
    }
}
```

The first execution of the script does not specify any switch parameters. The second execution directs the script to skip the first 5 objects.

```
1 [CmdletBinding(SupportsPaging = $true)]
2 param (
3     [parameter (ValueFromPipeline=$true, Mandatory=$true)] $num
4 )
5 begin { # initialize, happens once at the beginning
6     $sum = $nnum = $rawcnt = 0
7 }
8
9 process { # happens for each object in the pipeline
10    $rawcnt++
11
12    if ($rawcnt -gt $PSCmdlet.PagingParameters.Skip) {
13        $sum += $num
14        $nnum++
15    }
16
17 }
18
19 end { # happens once at the end
20     write-host "the sum is $sum"
21     if ($PSCmdlet.PagingParameters.IncludeTotalCount) {
22         write-host "$nnum numbers processed"
23     }
24
25 }
```

```
PS C:\Users>
PS C:\Users> 1..10 | .\sum-nums.ps1
the sum is 55

PS C:\Users> 1..10 | .\sum-nums.ps1 -skip 5
the sum is 40

PS C:\Users> 1..10 | .\sum-nums.ps1 -IncludeTotalCount
the sum is 55
10 numbers processed

PS C:\Users> 1..10 | .\sum-nums.ps1 -skip 5 -IncludeTotalCount
the sum is 40
5 numbers processed
```

Execution Policy

Sunday, February 24, 2013
8:13 AM

Given the historical vulnerabilities in previous Microsoft scripting languages, MS implemented a PowerShell execution policy to provide a more secure scripting environment. By default, the user cannot run scripts (Restricted) but can operate interactively in the PS window.

The execution policy is set by the cmdlet `Set-ExecutionPolicy` which must be run in administrative privileges:

```
Set-ExecutionPolicy [-ExecutionPolicy] <ExecutionPolicy> [[-Scope]
<ExecutionPolicyScope>] [-Force] [-Confirm]
```

Execution Policies:

Restricted (default): Does not load configuration files or run scripts.

AllSigned: All scripts and configuration files be signed by a trusted publisher, including scripts.

RemoteSigned: Requires that all scripts and configuration files downloaded from the Internet be signed by a trusted publisher. Allows scripts written on local computer to execute unrestricted.

Unrestricted: Loads all configuration files and runs all scripts. Unsigned script downloaded scripts prompt for permission to execute.

Bypass: Nothing is blocked and there are no warnings or prompts.

Undefined: Removes the currently assigned execution policy from the current scope. This parameter will not remove an execution policy that is set in a Group Policy scope.

Scope:

Process: The execution policy affects only the current Windows PowerShell process.

CurrentUser: The execution policy affects only the current user.

LocalMachine (default): The execution policy affects all users of the computer.

```
Set-ExecutionPolicy unrestricted -force  
# removes the unrestricted policy and assigns either the Group or  
Local Machine policy  
Set-ExecutionPolicy undefined -force
```

The *-force* switch parameter suppresses prompts.

Profiles scripts are automatically executed when PowerShell is executed. The examples below also show a prompt function. The purpose of a prompt function is to change the PowerShell prompt.

The profile script example sets the execution policy to all signed which means every script including the profile script must be digitally signed.

```
Set-ExecutionPolicy allsigned -force  
function prompt  
{  
    Write-Host ("PS " + $(get-location) + ">") -nonewline -  
    foregroundcolor gray  
}  
# SIG # Begin signature block  
# MIIEMwYJKoZIhvcNAQcCoIIEJDCCBACAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB  
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQABBAfzDtgwUsITrck0sYpfvNR  
# AgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQUqBmx3RCdfGZQFT7ycCpi13IZ  
# Trqgggi9MIICOTCCAagAwIBAgIQp9fTgAw2tIpOy4Z1DajSMjAJBgUrDgMCHQUA  
# MCwxKjAoBgNVBAMTIVBvd2VyU2h1bGwgTG9jYWwgQ2VydGlmawNhdGUgUm9vdDAe  
# Fw0xMTExMjYwMzI1MzFaFw0zOTEyMzEyMzU5NTlaMBoxGDAwBgnVBAMTD1Bvd2Vy  
# U2h1bGwgVXNlcjCBnzANBgkqhkiG9w0BAQEFAAOBJQAwgYkCgYEAt22/LQD6Ltb/  
# HosmWW+XR9MfuYTZE4w5C5qUN8tHQ5UJeohzlqRUV3CPiFAxohIwk0kzXyyk+cLU  
# 6uvx5m0mudeIVW7VxXZi+I4j/kb2YXc4uONEmUvT2hRV7e0kP1ZUpfe6WEcnVggP  
# zpxUstBL+4vz8NxFV403i+ALbdVzic0CAwEAAsN2MHQwEwYDVR01BAwwCgYIKwYB  
# BQUHAWMwXQYDVR0BBFYwVIAQIN9XpsV18RaksprD930GiqEuMCwxKjAoBgNVBAMT  
# IVBvd2VyU2h1bGwgTG9jYWwgQ2VydGlmawNhdGUgUm9vdIIQaZFkhnAtMrVEemk9
```

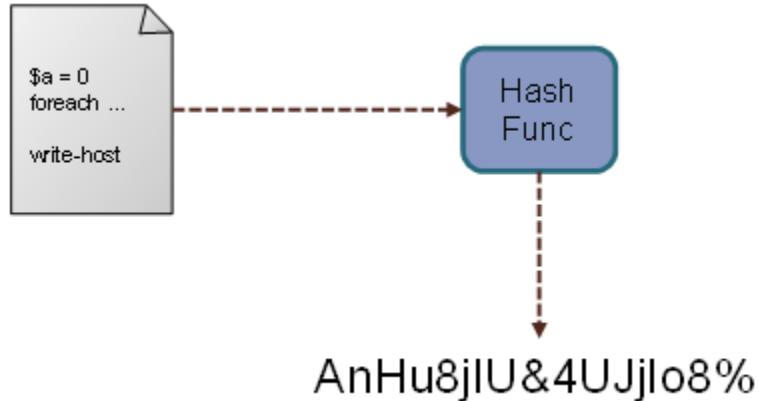
```
# WMRYIjAJBgUrDgMCHQUAA4GBAE0/0ZDkSRRhYzrVto/BIkyp6pobZWJw2q7wrJyF
# DoB7fRMaMo7aE0+5IrsT/BpsVpfSVEKgnxkgFkX9KfUwFbSRXNGcbLHnCCm32kMU
# 73hZ8FiV4hAsj6+xbUcfvfG/Sh/WqAm3BQnBCVq/sN3hS9Xd29xirN4Xgtqn50Aw
# X62iMYIBYDCCAVwCAQEwQDAsMSowKAYDVQQDEyFQb3d1c1NoZWxsIExvY2FsIEN1
# cnRpZm1jYXR1IFJvb3QCEKfx04AMNrSKTsuGdQ2o0jIwCQYFKw4DAhoFAKB4MBgG
# CisGAQQBgcjCAQwxCjAIoAKAAKEcgAAwGQYJKoZIhvcNAQkDMQwGCisGAQQBgcjCC
# AQQwHAYKKwYBBAGCNwIBCzEOMAwGCisGAQQBgcjCARUwIwYJKoZIhvcNAQkEMRYE
# FORjpJrLi32SW1BuToIiq2SNeRAtMA0GCSqGSIB3DQEBAQUABIGAbLCSh9AgauMg
# ySG9SblGt4+41IeICYzGcCUFXDlyJemvzoL2Iz0WhKXWMEU/juagkhvTi421LxK3
# yqkKDCjyuBlAp9dkcdAGJi9hd6tVVnELReuOc50WD+rthywdOBH4ZaDNE4vHyIiS
# R1smj5aujfbdXakChMPPTt0nSUrU6aw=
# SIG # End signature block
```

Digital Signatures

Monday, February 25, 2013
6:16 PM

An *digital signature* when applied confirms the authenticity of the executable and guarantees that the code has not been altered or corrupted since it was signed. Further, by affixing a digital signature to code, i.e., *code signing*, the author attests that the code functions as intended and does not deliberately harm the system on which it's used.

Signing code requires a *hash function*. A hash function reads each byte of the code and produces a unique string based upon the computational algorithm implemented by the hash function. The two functions commonly used are MD5 and SHA-1. SHA-1 is superior to MD5 and was the most common hash function used. SHA-2 which provides longer key lengths superseded SHA-1.



Code Signing:

For a script developer to sign a script, a *code signing certificate* must be used. The script writer has two choices a *self-signed certificate* or purchasing a certificate from a Certificate Authority like GoDaddy or GeoTrust.

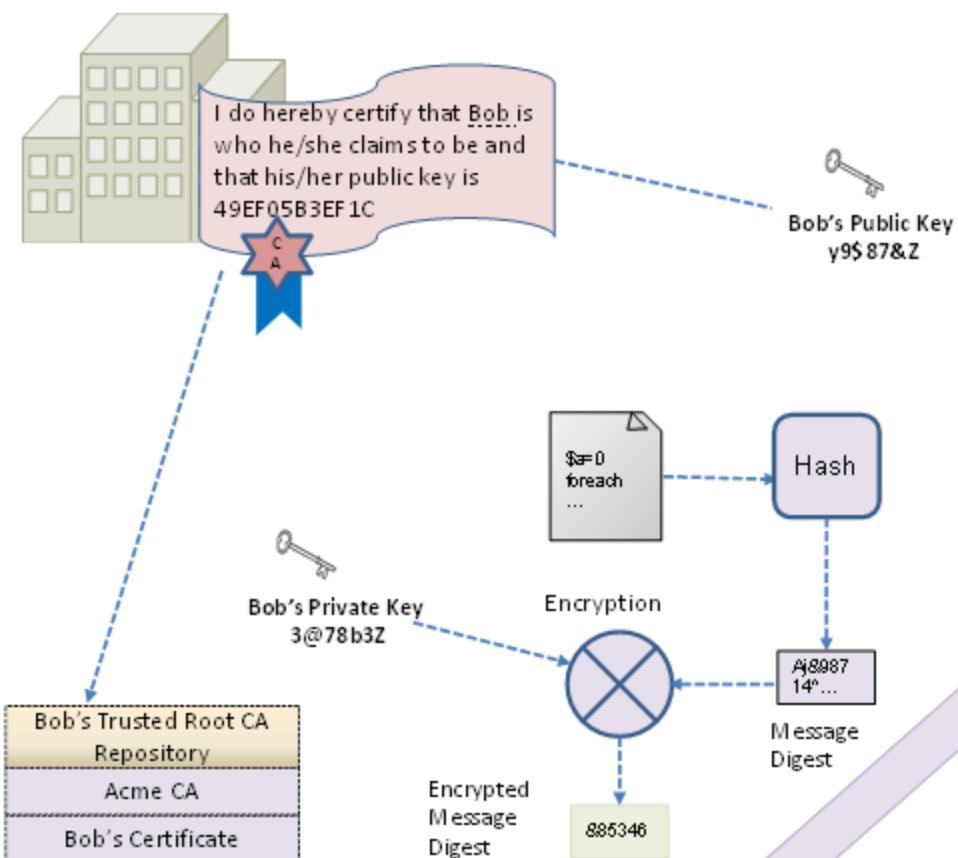
In the scenario below, Bob, the script developer, purchased a code signing certificate from Acme. The process of signing the script requires the script text to be *hashed*. The resulting hash is encrypted by Bob's private key that he received from the CA. The resulting encrypted hashed is added to the script.

When Alice downloads the Bob's script and opens the script in PowerShell, PowerShell does the following:

1. PowerShell verifies that Bob's certificate is from a *trusted CA*. In the example below Acme is trusted because Acme's certificate is in the Trusted Root or Intermediate CA reposition.
2. Once the signing authority is verified as a trusted source, the hash in the script is decrypted using Bob's public key.
3. PowerShell will then create a hash of the script code using the same algorithm as was used by Bob.
4. The calculated hash is compared against the decrypted hash. If they are the same, the script was not change and PowerShell executes the script. If the

hashes are different, the script was changed after it was signed and cannot be trusted. In the case, PowerShell will not execute the script and throw an error indicating the problem.

Acme CA



Acme's cert
my repository
get Bob's pu

\$a=0
foreach
...
886346
Acme
CA

Bob's Public Ke
y9\$87&Z

Decryption

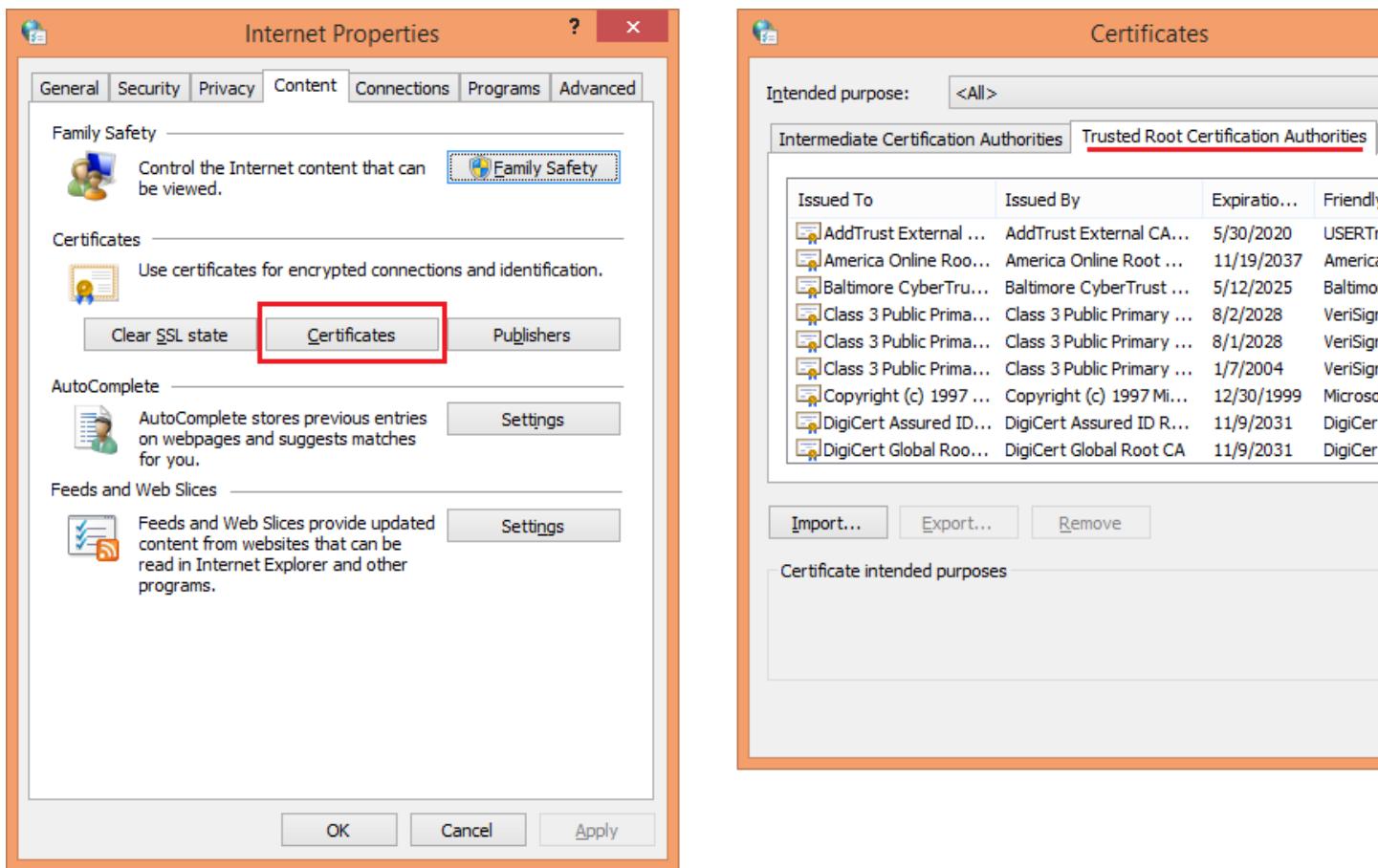


Bob the Developer

This code can't
be trusted
because it
change since
Bob signed it..

Trusted CAs:

Windows maintains a couple of repositories for trusted certificates. They may be viewed by navigating to the Control Panel then open Internet Options. In the Internet Options window, select the Content tab the click them Certificates button.



Any certificate signed by any of the authorities listed in the Trusted Root Certification Authorities pane are trusted by Windows. If a certificate is signed by an Intermediate Certification Authority, the complete certificate chain is verified. This means that the root CA must be in the trusted repository for the certificate to be acceptable.

Signing Scripts

Sunday, February 24, 2013
1:29 PM

The steps below allow you to create a local CA that can issue personal certificates. This is strictly intended for testing purposes and not for production. Scripts signed using the steps below may only execute on the computer where the local CA resides. They will not execute on another computer because the certificate chain cannot be built. If this is a requirement, a personal certificate must be purchased from a trusted CA and installed on your computer.

The makecert tool creates Local Certificate Authority which allows you to create X509 certificates that may be used to sign scripts.

X509 certificates use a private and public key.

Download makecert.zip from Technet site:

<http://gallery.technet.microsoft.com/Certificate-Creation-tool-5b7c054d>

Extract and copy makecert.exe to a convenient location.

See [http://msdn.microsoft.com/en-us/library/windows/desktop/aa386968\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa386968(v=vs.85).aspx)

Create the Local Certificate Authority:

Since this updates the registry, the cmd or PowerShell must be run with Administrative privileges.

```
makecert -n "CN=CIS161" -a sha1 -eku 1.3.6.1.5.5.7.3.3 -r -sv  
powershell.pvk -ss Root -sr localMachine powershell.cer
```

- n Name of the publisher certificate
- a Algorithm used to generate the signature
- eku Enhanced key usage object identifiers eku OID specifies how the certificate will be used
- r Create self-signed cert
- sv File to store the private key
- ss The name of the certificate store where the cert is stored
- sr Registry location of the certificate store: LocalMachine (HKEY_LOCAL_MACHINE) or CurrentUser (HKEY_CURRENT_USER)

The above creates the certificate for the Local Certificate Authority. This may be viewed by navigating to: Control Panel ⇒ Internet Options ⇒ Content tab ⇒ Certificates button ⇒ Trusted Root Certificate Authorities

Copy powershell.pvk and powershell.cer to safe location. In the example below the files are resident in the LocalCA directory.

Create a personal code signing certificate:

```
makecert -pe -n "CN=Student" -ss MY -a sha1 -eku 1.3.6.1.5.5.7.3.3 -  
iv .\LocalCA\powershell.pvk -ic .\LocalCA\powershell.cer
```

The above creates the personal certificate for code signing purposes. This may be viewed by navigating to: Control Panel ⇒ Internet Options ⇒ Content tab ⇒ Certificates button ⇒ Personal

Verification of the proper installation of the certification may also be done in PS:

```
Get-ChildItem cert:\ -recurse -codesign
```

Note: the directory specifies "CurrentUser" since this is a personal certificate.

Signing a script:

In the directory where the script is located, execute:

```
Set-AuthenticodeSignature hello2.ps1 $( Get-ChildItem cert:\CurrentUser\My -codesign )
```

Edit the script view the signature.

Right-click on the script then select "Properties". Click the "Digital Signatures" tab to view info on cert.

Examining the Certificate in PowerShell:

```
# get the certificate(s)
$cert = gci cert:\CurrentUser\My -codesign

$cert | gm

#view the information
$cert.IssuerName
$cert.Path

# is the certificate trusted
$cert.verify()

# load the security assembly
[System.Reflection.Assembly]::LoadWithPartialName("System.Security")

# display the cert
[System.Security.Cryptography.X509Certificates.X509Certificate2UI]::DisplayCertif
```

Exporting the Root Certificate:

```

# get the certificate
$rootCert = gci Cert:\LocalMachine\root | ? { $_.Subject -like
'*CIS161*' }

# export it as hex
getBytes = $rootCert.Export("cert")

# write to the a file to the home directory
[System.IO.File]::WriteAllBytes("$home\myroot.cer", $bytes)

```

The certificate may now be copied to target computer and installed in the Trusted Root Authorities certificate store. Doing this allows scripts signed with this certificate to execute on the remote computer.

Background Jobs

Saturday, April 26, 2014
10:47 AM

Two cmdlets may be used to start jobs: Start-Job and Invoke-Command. Start-Job starts a background job that runs entirely on the local computer and typically doesn't use the remoting subsystem. Invoke-Command starts a job that's tracked on local machine but sends commands to remote computers for execution. Invoke-Command is useful in running a script on multiple hosts.

Note: If Invoke-Command is used locally with the -Computername or -AsJob parameter, PowerShell will use Remoting to the local computer, so it must be enabled.

The cmdlets used in background processing are:

Get-Job	List the active and inactive jobs.
Start-Job	Start a job in the background
Wait-Job	Wait for a job to complete
Stop-Job	Stops a current executing job
Receive-Job	Receive the results of a job

```
Remove-Job
```

```
Remove an job
```

Working with Jobs

Monday, April 28, 2014
11:04 AM

In this script the test-connection is executed in the background. Since the background script is executing in a separate runspace, the results of the test-connection are not communicated back to the foreground script that started it.

```
start-job -ScriptBlock { test-connection 8.8.8.8 -count 1}
```

```
PS C:\Users> start-job -ScriptBlock { test-connection 8.8.8.8 -count 1}
Id     Name          PSJobTypeName   State      HasMoreData  Location
--     --           -----          -----      -----       -----
2      Job2          BackgroundJob  Running    True        LocalHost

```

Many cmdlets have the -AsJob switch available. This parameter is the equivalent of executing the cmdlet with start-job.

```
PS C:\Users> test-connection 8.8.8.8 -count 1 -AsJob
Id     Name          PSJobTypeName   State      HasMoreData  Location
--     --           -----          -----      -----       -----
12     Job12         WmiJob        Running    True        .

```

Ideally, we would like to communicate information from the job back to the caller environment. We might be tempted to try something like the script below.

```
$result = ''
start-job -ScriptBlock { $result = test-connection 8.8.8.8 -count 1}
write-host $result

$result = test-connection 8.8.8.8 -count 1 -AsJob
```

```
$result
```

```
PS C:\Users>
PS C:\Users> $result = ''
PS C:\Users> start-job -ScriptBlock { $result = test-connection 8.8.8.8 -count 1}
Id      Name          PSJobTypeName   State    HasMoreData Location
--      --           BackgroundJob   Running   True       localhost
20      Job20         BackgroundJob   Running   True       localhost

PS C:\Users> write-host $result
PS C:\Users>
PS C:\Users> $result = test-connection 8.8.8.8 -count 1 -AsJob
PS C:\Users>
PS C:\Users> $result
Id      Name          PSJobTypeName   State    HasMoreData Location
--      --           WmiJob        Completed True       .
22      Job22         WmiJob        Completed True       .
```

This doesn't work because the code in the script block runs in a separate PowerShell runspace and variables in one runspace are not normally available in another. Note, there are methods of inter-runspace communications but not when using PowerShell jobs.

The only option for receiving information from a background job is through the Receive-Job cmdlet. The Start-Job cmdlet returns a PSRemotingJob object. This object is used to receive results back to the calling environment.

```
$job = start-job -ScriptBlock { test-connection 8.8.8.8 -count 1}
$job
receive-job $job

$job = test-connection 8.8.8.8 -count 1 -AsJob
receive-job $job
```

```

PS C:\Users> $job = start-job -ScriptBlock { test-connection 8.8.8.8 -count 1}
PS C:\Users> $job
Id      Name          PSJobTypeName      State       HasMoreData      Location
--      --          BackgroundJob      Completed    True           localhost
36      Job36        BackgroundJob      Completed    True           localhost

PS C:\Users> receive-job $job
Source      Destination      IPV4Address      IPV6Address
-----      -----          -----          -----
EL-CID      8.8.8.8          8.8.8.8

PS C:\Users> $job = test-connection 8.8.8.8 -count 1 -AsJob
PS C:\Users> $job
Id      Name          PSJobTypeName      State       HasMoreData      Location
--      --          WmiJob          Completed    True           .
38      Job38        WmiJob          Completed    True           .

PS C:\Users> receive-job $job
Source      Destination      IPV4Address      IPV6Address
-----      -----          -----          -----
EL-CID      8.8.8.8          8.8.8.8

```

One of the issues that when using Receive-Job is that all cmdlet output normally emitted to the console is returned by Receive-Job. This example illustrates this issue. Each cmdlet in the script block emits output to the console. This output is cached and returned when Receive-Job is executed and receives the results.

```

$job = start-job -ScriptBlock {
    test-connection 8.8.8.8 -count 1
    get-process | select-object -first 2
    get-service | select object -first 2
}
receive-job $job

```

```

PS C:\Users>
PS C:\Users> $job = start-job -ScriptBlock {
>>>         test-connection 8.8.8.8 -count 1
>>>         get-process | select-object -first 2
>>>         get-service | select-object -first 2
>>>
PS C:\Users> receive-job $job
PS C:\Users>
PS C:\Users> receive-job $job
Nothing received since
job is still executing

Source          Destination        IPV4Address      IPV6Address
-----          -----           8.8.8.8
EL-CID          8.8.8.8          8.8.8.8

Id      : 8932
Handles : 332
CPU     : 3.875
Name    : Amazon Music Helper

Id      : 10912
Handles : 852
CPU     : 19.421875
Name    : ApplicationFrameHost

object   :
RunspaceId : 088334d9-d366-4d4d-972a-d2a54db091c4

object   :
RunspaceId : 088334d9-d366-4d4d-972a-d2a54db091c4

```

The above also illustrates another issue. The first execution of Receive-Job returned nothing since the background job is still executing. To remediate these issues we will control what is returned to the calling environment by storing only what is to be returned and discarding everything else. We will also use the Wait-Job cmdlet to wait until the job completes before executing Receive-Job.

In the script below, The output of test-connection is saved so it may be returned later. We use Out-Null to discard the output of the get-process pipeline. The [void] type with the get-service sub-expression also discards the out.

```

$job = start-job -ScriptBlock {
    $result = test-connection 8.8.8.8 -count 1
    get-process | select-object -first 2 | out-null
    [void] (get-service | select-object -first 2)
    start-sleep -seconds 5
    return $result
}
wait-job $Job
receive-job $job

```

```

PS C:\Users>
PS C:\Users> $job = start-job -ScriptBlock {
>>>     $result = test-connection 8.8.8.8 -count 1
>>>     get-process | select-object -first 2 | out-null
>>>     [void] (get-service | select-object -first 2)
>>>     return $result
>>>
PS C:\Users> wait-job $Job
Id      Name          PSJobTypeName   State    HasMoreData  Location
--      --          BackgroundJob   Completed  True        localhost
42      Job42

PS C:\Users> receive-job $job
Source      Destination    IPV4Address    IPV6Address
-----      -----          -----          -----
EL-CID      8.8.8.8       8.8.8.8

```

As seen above knowing when a background job completes is useful. The Get-Job cmdlet is useful in providing information about active and inactive jobs.

```

start-job -ScriptBlock { test-connection 8.8.8.8 -count 1 }
start-job -ScriptBlock {
    start-sleep -seconds 10
    test-connection 8.8.8.8 -count 1
}

get-job

get-job

get-job

```

PS C:\Users>	start-job -ScriptBlock { test-connection 8.8.8.8 -count 1 }						
Id	Name	PSJobTypeName	State	HasMoreData	Location		Command
--	--	--	--	--	--		
2	Job2	BackgroundJob	Running	True	localhost		test
PS C:\Users>	start-job -ScriptBlock { start-sleep -seconds 10 test-connection 8.8.8.8 -count 1 }						
Id	Name	PSJobTypeName	State	HasMoreData	Location		Command
--	--	--	--	--	--		
4	Job4	BackgroundJob	Running	True	localhost		...
PS C:\Users>	get-job						
Id	Name	PSJobTypeName	State	HasMoreData	Location		Command
--	--	--	--	--	--		
2	Job2	BackgroundJob	Running	True	localhost		test
4	Job4	BackgroundJob	Running	True	localhost		...
PS C:\Users>	get-job						
Id	Name	PSJobTypeName	State	HasMoreData	Location		Command
--	--	--	--	--	--		
2	Job2	BackgroundJob	Running	True	localhost		test
4	Job4	BackgroundJob	Running	True	localhost		...
PS C:\Users>	get-job						
Id	Name	PSJobTypeName	State	HasMoreData	Location		Command
--	--	--	--	--	--		
2	Job2	BackgroundJob	Completed	True	localhost		test
4	Job4	BackgroundJob	Running	True	localhost		...

As is evident from the above example, when multiple jobs are executed identifying which job is which is difficult. The -Name parameter is useful in this case since we may apply descriptive identifiers to the job.

```
start-job -Name 'Ping Google' -ScriptBlock {
    test-connection 8.8.8.8 -count 5
}
start-job -Name 'Ping HUT' -ScriptBlock {
    test-connection 130.233.224.254 -count 5
}
start-job -Name 'Ping WCC' -ScriptBlock {
    test-connection 207.75.134.1 -count 5
}

get-job -Name 'Ping HUT'
```

```
get-job -Name 'Ping WCC'
```

```
get-job
```

```
PS C:\Users>
PS C:\Users> start-job -Name 'Ping Google' -ScriptBlock {
>>>     test-connection 8.8.8.8 -count 5
>>> }
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
6	Ping Google	BackgroundJob	Running	True	localhost

```
PS C:\Users> start-job -Name 'Ping HUT' -ScriptBlock {
>>>     test-connection 130.233.224.254 -count 5
>>> }
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
8	Ping HUT	BackgroundJob	Running	True	localhost

```
PS C:\Users> start-job -Name 'Ping WCC' -ScriptBlock {
>>>     test-connection 207.75.134.1 -count 5
>>> }
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
10	Ping WCC	BackgroundJob	Running	True	localhost

```
PS C:\Users>
PS C:\Users> get-job -Name 'Ping HUT'
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
8	Ping HUT	BackgroundJob	Running	True	localhost

```
PS C:\Users>
PS C:\Users> get-job -Name 'Ping WCC'
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
10	Ping WCC	BackgroundJob	Running	True	localhost

```
PS C:\Users>
PS C:\Users> get-job
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
2	Job2	BackgroundJob	Completed	True	localhost
4	Job4	BackgroundJob	Completed	True	localhost
6	Ping Google	BackgroundJob	Running	True	localhost
8	Ping HUT	BackgroundJob	Running	True	localhost
10	Ping WCC	BackgroundJob	Running	True	localhost

Script files may also be used in Start-Job. Assuming the script pings.ps1 contains the following:

```
test-connection 8.8.8.8 -count 5  
test-connection 130.233.224.254 -count 5  
test-connection 207.75.134.1 -count 5
```

Then the syntax below is used to run the script in the background:

```
start-job -FilePath pings.ps1 -Name 'Pinger'  
get-job
```

```
PS C:\Users>  
PS C:\Users> start-job -FilePath pings.ps1 -Name 'Pinger'  


| Id | Name   | PSJobTypeName | State   | HasMoreData | Location  |
|----|--------|---------------|---------|-------------|-----------|
| 14 | Pinger | BackgroundJob | Running | True        | localhost |

  
PS C:\Users>  
PS C:\Users> get-job  


| Id | Name        | PSJobTypeName | State     | HasMoreData | Location  |
|----|-------------|---------------|-----------|-------------|-----------|
| 2  | Job2        | BackgroundJob | Completed | True        | localhost |
| 4  | Job4        | BackgroundJob | Completed | True        | localhost |
| 6  | Ping Google | BackgroundJob | Completed | True        | localhost |
| 8  | Ping HUT    | BackgroundJob | Completed | True        | localhost |
| 10 | Ping WCC    | BackgroundJob | Completed | True        | localhost |
| 12 | Pinger      | BackgroundJob | Running   | True        | localhost |
| 14 | Pinger      | BackgroundJob | Running   | True        | localhost |

  
PS C:\Users> get-job 'Pinger'  


| Id | Name   | PSJobTypeName | State   | HasMoreData | Location  |
|----|--------|---------------|---------|-------------|-----------|
| 12 | Pinger | BackgroundJob | Running | True        | localhost |
| 14 | Pinger | BackgroundJob | Running | True        | localhost |

  
PS C:\Users> get-job 'Pinger'  


| Id | Name   | PSJobTypeName | State   | HasMoreData | Location  |
|----|--------|---------------|---------|-------------|-----------|
| 12 | Pinger | BackgroundJob | Running | True        | localhost |
| 14 | Pinger | BackgroundJob | Running | True        | localhost |


```

WinRM Overview

Wednesday, November 20, 2013
9:35 PM

Remoting refers to the ability to execute PowerShell commands and scripts on remote computers.

Classic remoting, such as WMI, in PowerShell typically used protocols such as DCOM and RPC to make connections to remote machines. Using these protocols requires opening a number of ports in the firewall.

Starting with PowerShell 3.0 the WS-MAN (Web-Services Management) protocol, using only port 5985, and WinRM (Windows Remote Management) service are used to implement remoting.

Starting with Server 2012 PowerShell remoting is enable automatically. Remoting must be enabled on both the target host and the client host. This means that when connecting to a target host that has remoting enabled, the client host must also have remoting enabled.

Enabling Remoting

Thursday, November 21, 2013
7:34 AM

Windows Server 2012 installs with Windows Remote Management (WinRM) configured and running. The default installation of Windows 7 or 8 hosts includes Windows Remote Management; however, remote management needs to be enabled.

Since WinRM requires reciprocal communications, remoting must be enabled on all hosts engaging remote management including the host from which the remote session is initiated.

PowerShell provides a convenient method of enabling remoting through the command shell using winrm quickconfig or through PowerShell using the Enable-PSRemoting. Both do the following:

1. Starts or restarts the WinRM service

2. Sets the WinRM service startup type to Automatic
3. Creates a listener to accept requests from any Internet Protocol (IP) address
4. Enables inbound firewall exceptions for WSMAN traffic
5. Sets a target listener named Microsoft.powershell
6. Sets a target listener named Microsoft.powershell.workflow
7. Sets a target listener named Microsoft.powershell32

You need to execute the PowerShell console with elevated privileges. Also, the network interfaces must be set to Private or Domain and not Public. If the network interface is set to Public, an error is thrown when step 4, enable firewall exceptions, is attempted.

```
PS C:\user>
PS C:\user> Enable-PSRemoting

WinRM Quick Configuration
Running command "Set-WSManQuickConfig" to enable remote management of this computer by using the
This includes:
  1. Starting or restarting (if already started) the WinRM service
  2. Setting the WinRM service startup type to Automatic
  3. Creating a listener to accept requests on any IP address
  4. Enabling Windows Firewall inbound rule exceptions for WS-Management traffic (for http only

Do you want to continue?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
WinRM is already set up to receive requests on this computer.
WinRM is already set up for remote management on this computer.

Confirm
Are you sure you want to perform this action?
Performing the operation "Set-PSSessionConfiguration" on target "Name: microsoft.powershell SDDL:
This lets selected users remotely run Windows PowerShell commands on this computer."
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y

Confirm
Are you sure you want to perform this action?
Performing the operation "Set-PSSessionConfiguration" on target "Name: microsoft.powershell.workflow SDDL:
This lets selected users remotely run Windows PowerShell commands on this computer."
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y

Confirm
Are you sure you want to perform this action?
Performing the operation "Set-PSSessionConfiguration" on target "Name: microsoft.powershell32 SDDL:
This lets selected users remotely run Windows PowerShell commands on this computer."
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
```

The cmdlet Test-WSMan may be used to determine if WinRM is configured and running on the remote computer:

```
test-wsman -computername 10.10.10.100
```

```
PS C:\Users>
PS C:\Users> test-wsman -computername 10.10.10.100

wsmid      : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor  : Microsoft Corporation
ProductVersion : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

Configuration in a Domain Network

For Server 2012, the above scripts to enable PowerShell remoting may not complete the configuration. After executing Enable-PSRemoting or winrm quickconfig on the server to be managed, open the System event log and look for a warning with Event ID 10154 with the message "The WinRM service failed to create the following SPN: %1. " The Service Provider Name (SPN) is not created by default since this may constitute a security vulnerability.

To create the SPN so that PowerShell remoting works, enter the commands in an elevated PowerShell console.

```
setspn -f -S WSMAN/<domain> <server name>
setspn -f -S WSMAN/<server> <server name>
```

Where <server> is the name of the server to be managed and <domain> is the Active Directory domain.

Host Authentication

Thursday, November 21, 2013
7:35 AM

WinRM implements two levels of authentication: user and machine. User-level authentication delegates the user credentials to the remote machine. The remote machine will do only the tasks permitted for that identity.

Machine level authentication implies that the remote host *trusts* the host from which the PowerShell commands (scripts) are being issued. The remote host will refuse connections originating from untrusted hosts. Trust is mutual, this means that both hosts must trust each other.

In the exam below, we attempt to establish a remote session to 10.10.10.100

```
Enter-PsSession 10.10.10.100
```

```
PS C:\Users>
PS C:\Users> Enter-PSSession 10.10.10.100
Enter-PSSession : Connecting to remote server 10.10.10.100 failed with the following error:
cannot process the request. If the authentication scheme is different from Kerberos, make sure the
joined to a domain, then HTTPS transport must be used or the destination machine must have a valid
configuration setting. Use winrm.cmd to configure TrustedHosts. Note that computers must be
authenticated. You can get more information about that by running the following command. For more
information, see the about_Remote_Troubleshooting Help topic.
At line:1 char:1
+ Enter-PSSession 10.10.10.100
+ ~~~~~
+ CategoryInfo          : InvalidArgument: (10.10.10.100:String) [Enter-PSSession]
+ FullyQualifiedErrorId : CreateRemoteRunspaceFailed
```

Trust may be accomplished through mutual membership in Active Directory. If the one or both hosts are not a member of Active Directory, then mutual trust may be established by adding the remote host to the WinRM TrustedHosts store.

Trust may be established using winrm.cmd in a command shell.

```
winrm set winrm/config/Client @{TrustedHosts="10.10.10.100"}
```

or PowerShell.

```
set-item WSMAN:\localhost\client\TrustedHosts "10.10.10.100" -force
```

In the above examples, the IP address shown is that of the remote host. This command must be executed on both the host doing the managing and the host being managed. When the command is executed on the host doing the managing, the IP address must be that of the host being managed. Similarly, when the command is executed on the host being managed, the IP address must be that of the host doing the managing.

Example:

Given that the administrator on host 10.10.10.100 is managing the server identified by 10.10.10.200, the administrator must do the following:

On host 10.10.10.100:

```
set-item WSMAN:\localhost\client\TrustedHosts "10.10.10.200" -force
```

On host 10.10.10.200:

```
set-item WSMAN:\localhost\client\TrustedHosts "10.10.10.100" -force
```

Host Authentication in A Domain

Since PowerShell remoting may constitute a security vulnerability, host authentication is not enabled by default. To enable host authentication, run gpedit and navigate to Computer Configuration -> Administrative Templates -> Windows Components -> Windows Remote Management (WinRM) -> WinRM Service. Double-click "Allow remote server management through WinRM".

Local Group Policy Editor

File Action View Help

Sync your settings

Tablet PC

Task Scheduler

Windows Calendar

Windows Color System

Windows Customer Experience Improvement Program

Windows Defender

Windows Error Reporting

Windows Installer

Windows Logon Options

Windows Mail

Windows Media Center

Windows Media Digital Rights Management

Windows Media Player

Windows Messenger

Windows Mobility Center

Windows PowerShell

Windows Reliability Analysis

Windows Remote Management (WinRM)

WinRM Client

WinRM Service

Windows Remote Shell

Windows SideShow

Windows System Resource Manager

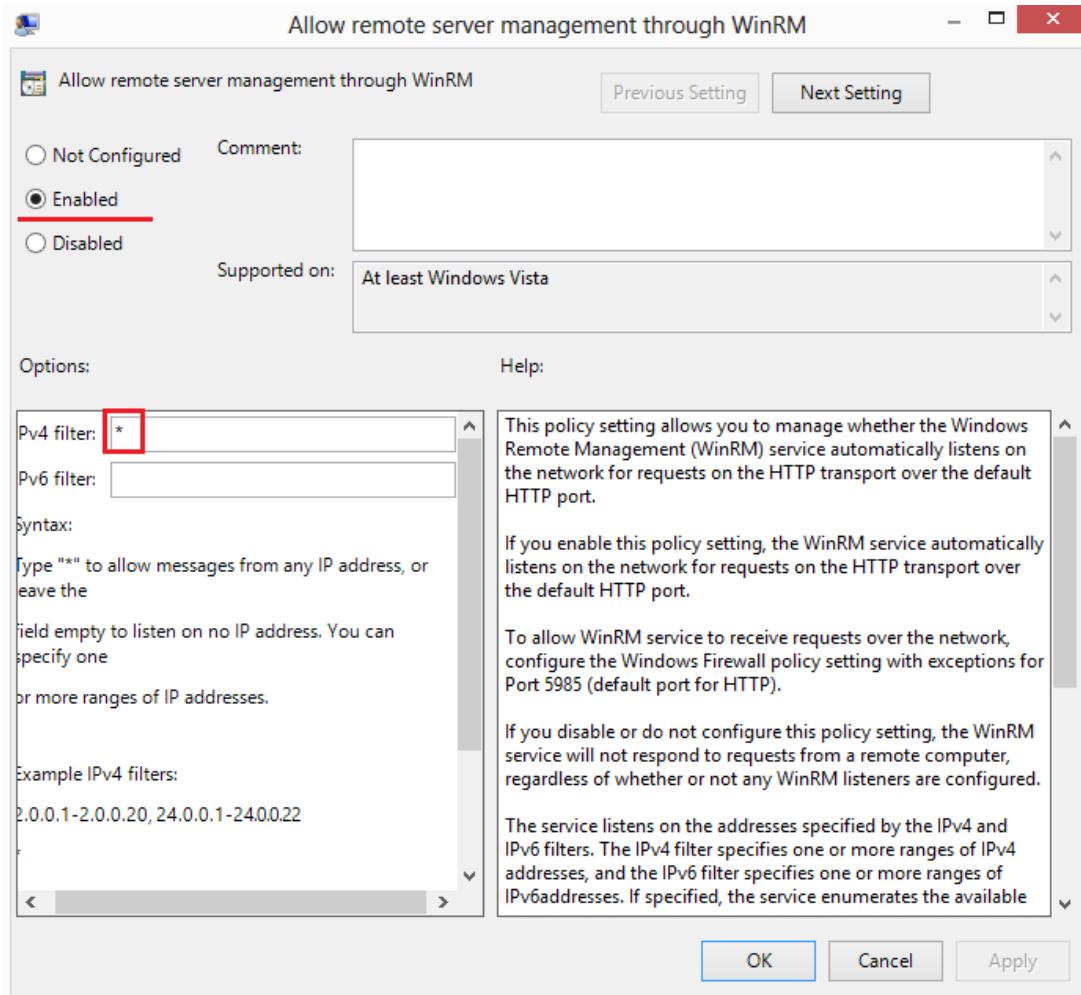
Windows Update

All Settings

User Configuration

Setting	State	Comment
Allow remote server management through WinRM	Enabled	
Allow Basic authentication	Not configured	
Allow CredSSP authentication	Not configured	
Allow unencrypted traffic	Not configured	
Specify channel binding token hardening level	Not configured	
Disallow WinRM from storing RunAs credentials	Not configured	
Disallow Kerberos authentication	Not configured	
Disallow Negotiate authentication	Not configured	
Turn On Compatibility HTTP Listener	Not configured	
Turn On Compatibility HTTPS Listener	Not configured	

Click the Enabled radio button then enter * on the IPv4 filter. The wildcard allows all subnets. If remoting to the server needs to be restricted enter an appropriate filter as described.



Click OK to close the dialog box. Close the gpedit windows the execute gpupdate to update the policy.

Remote Sessions

Thursday, November 21, 2013
7:35 AM

In addition to host authentication, the user attempting to establish a remote session must also be authenticated. If both hosts are part of the AD domain, and the administrator is signed into the domain no additional authentication is necessary. In a non-AD environment if the user attempting the remote session has

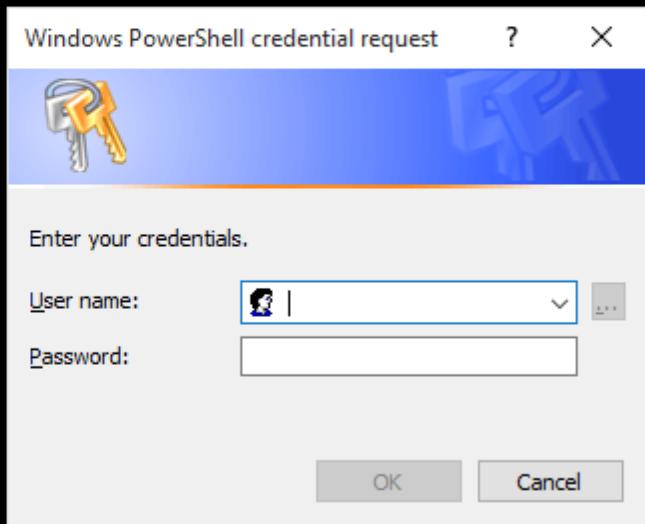
identical credentials the authentication occurs automatically. In any other case, the user must present authentication credentials to the remote host.

```
Enter-PSSession -ComputerName 10.10.10.100
```

In the case where the administrator is not signed into the domain, the administrator must present credentials before the session is established.

```
$cred = Get-Credential  
Enter-PSSession -ComputerName 10.10.10.100 -Credential $cred
```

```
PS C:\Users>  
PS C:\Users> $cred = Get-Credential  
cmdlet Get-Credential at command pipeline position 1  
Supply values for the following parameters:  
Credential
```



```

PS C:\Users>
PS C:\Users> $cred = Get-Credential

cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
PS C:\Users> Enter-PSSession -ComputerName 10.10.10.100 -Credential $cred
[10.10.10.100]: PS C:\Users\mgalea\Documents> $l ..
[10.10.10.100]: PS C:\Users\mgalea> $l ..
[10.10.10.100]: PS C:\Users>
[10.10.10.100]: PS C:\Users> get-process

```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	ID	ProcessName
209	11	1336	2824	43	20.31	364	csrss
119	10	1380	7636	44	3.78	428	csrss
154	12	1560	21916	182	5.02	3116	csrss
327	31	45808	47972	906	250.53	1440	dfsrs
121	12	2096	4728	34	0.13	1584	dfssvc
88	7	1116	4080	53	0.02	3868	dllhost
5223	3360	48488	40136	95	87.23	1468	dns
156	15	10516	46304	135	1.20	708	dwm
191	16	19820	29332	116	2.72	864	dwm
932	51	17392	42400	379	7.69	2992	explorer
976	49	19520	54572	393	9.98	3504	explorer
0	0	0	20	0		0	Idle
98	11	1720	3752	27	0.25	1492	ismserv
1695	115	24004	34120	1438	3,685.23	536	lsass
671	45	43848	39892	587	5.03	1400	Microsoft.ActiveDirectory.LDAP
154	17	2788	5476	61	0.09	2404	msdtc
213	11	1804	7092	86	0.42	2672	rdpclip
444	48	99836	32352	756	1,325.19	888	ServerManager
398	41	90136	25580	741	1,381.59	3336	ServerManager
281	13	5472	9296	292	803.64	528	services
53	3	300	764	5	0.05	272	smss
452	25	5872	14168	99	4.11	1372	spoolsv
236	15	2612	4748	45	125.56	52	svchost
671	50	12696	18156	1390	24.48	312	svchost
442	29	13868	17272	902	20.80	592	svchost
508	24	7244	13988	128	21.59	712	svchost
382	13	3100	7712	36	42.72	720	svchost
366	18	3688	5732	28	127.78	764	svchost
364	33	11048	12144	61	1.34	816	svchost
402	20	15280	17056	55	7,210.52	876	svchost

After the session is entered, any commands entered in the PowerShell console are executed on the remote hosts. The console prompt changes to indicate the remote host on which the session is executing.

The exit command exits the remote session and returns control back to the host on which the session was started.

Another method for establishing remote session is the New-PSSession cmdlet which creates a PSSession object. This is useful if the commands to execute on the remote host(s) are to be dynamically determined or if multiple remote sessions need to be established.

```
$remhost = new-PSSession -ComputerName 10.10.10.100 -Credential $cred
```

```
PS C:\Users> $remhost = new-PSSession -ComputerName 10.10.10.100 -Credential $cred
PS C:\Users> $remhost
Id Name ComputerName State ConfigurationName Availability
-- -- -- -- -- --
2 Session2 10.10.10.100 Opened Microsoft.PowerShell Available
```

The Invoke-Command cmdlet may be used with the PSSession object to execute commands on the remote host.

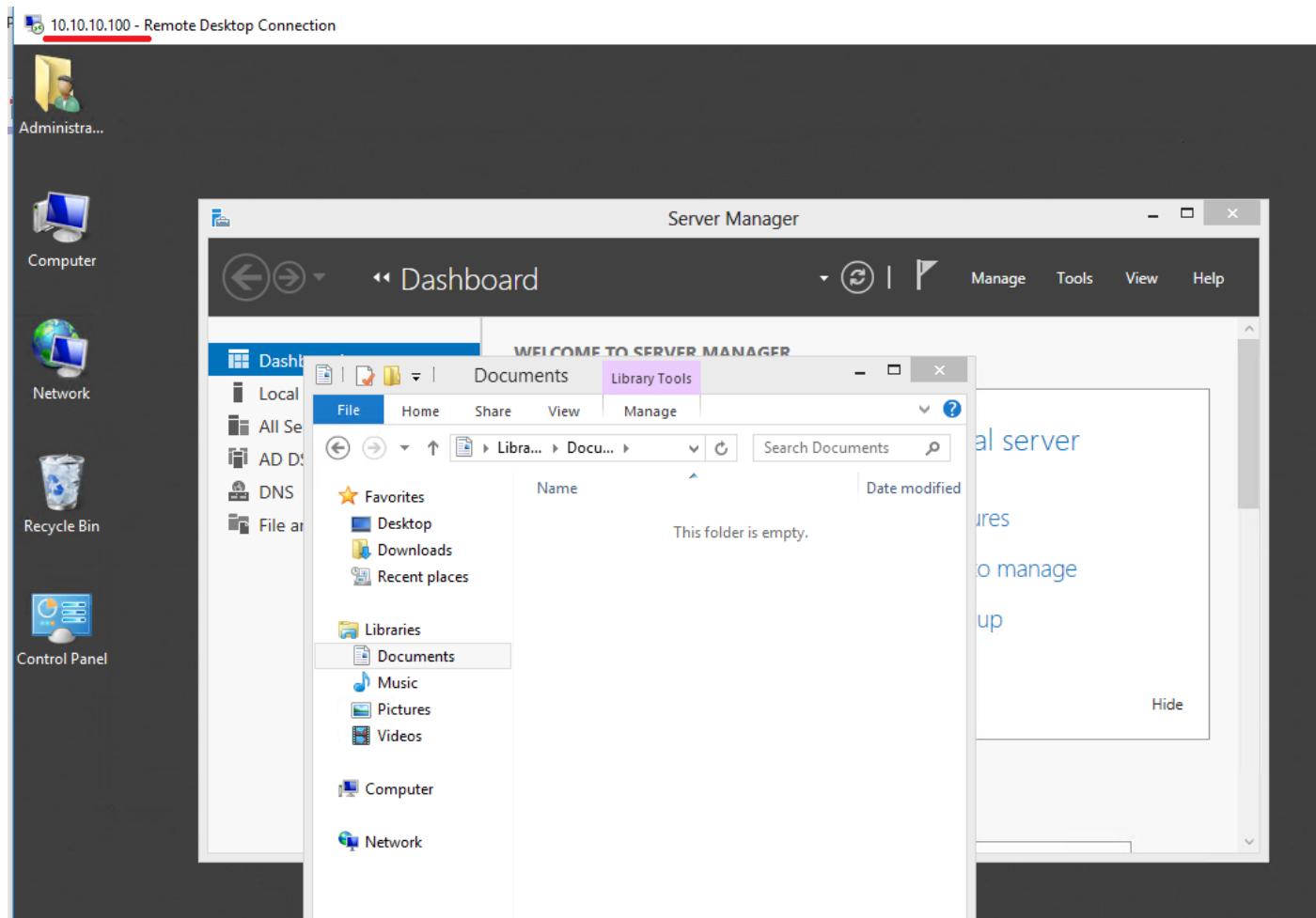
```
Invoke-Command -Session $remhost -ScriptBlock {get-process}
```

```
PS C:\Users> $remhost = new-PSSession -ComputerName 10.10.10.100 -Credential $cred
PS C:\Users> $remhost
Id Name ComputerName State ConfigurationName Availability
-- -- -- -- -- --
2 Session2 10.10.10.100 Opened Microsoft.PowerShell Available

PS C:\Users> Invoke-Command -Session $remhost -ScriptBlock {get-process}
Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName
-- -- -- -- -- --
210 11 1336 2824 43 20.31 364 csrss
119 10 1380 7636 44 3.78 428 csrss
154 12 1560 21916 182 5.02 3116 csrss
329 31 46020 48060 906 250.55 1440 dfssrs
121 12 2096 4728 34 0.13 1584 dfssvc
88 7 1116 4080 53 0.02 3868 dllhost
5221 3360 48764 40152 96 87.25 1468 dns
156 15 10516 46304 135 1.20 708 dwm
191 16 19820 29332 116 2.72 864 dwm
929 51 17392 42400 379 7.69 2992 explorer
976 49 19520 54572 393 9.98 3504 explorer
0 0 0 20 0 0 Idle
98 11 1720 3752 27 0.25 1492 ismserv
1685 114 24012 34160 1438 3,685.41 536 lsass
712 45 43840 39884 587 5.03 1400 Microsoft.ActiveDirectory.WebSer...
154 17 2788 5476 61 0.09 2404 msdtc
213 11 1804 7092 86 0.42 2672 rdpclip
438 47 99808 3604 755 1 325 19 888 ServerManager
PS Computer
```

Once a session is established, files may be copied to or from the remote host using the session. The image below shows a RDP window for 10.10.10.100 host

before the session is established. Note, that in the before image below no files are present in the Documents directory of the remote host.

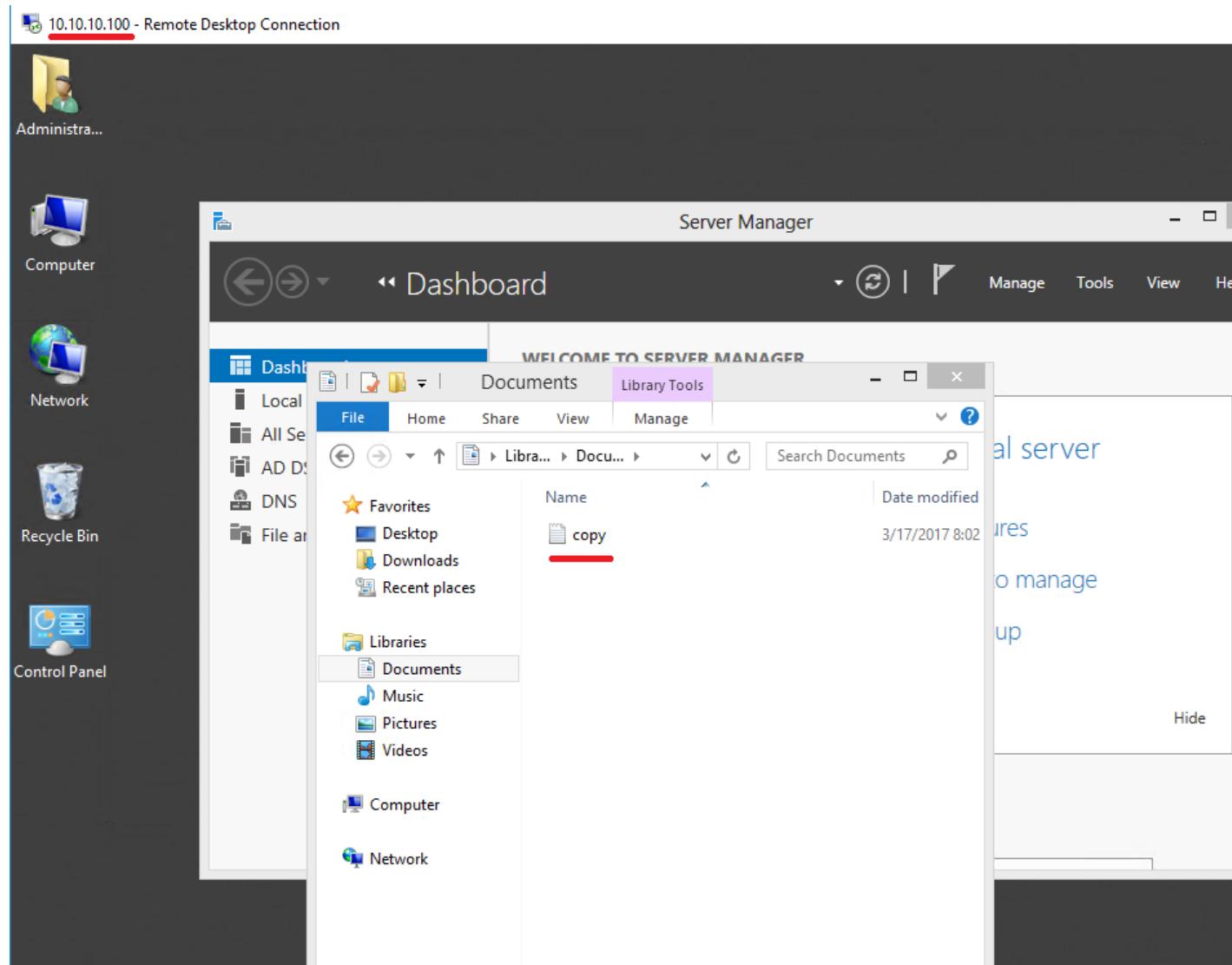


The cmdlet below copies the local file d:\copy.log to the remote host to which the session is connected.

```
copy-item d:\copy.log -destination "c:\users\administrator\documents"  
-tosession $remhost
```

```
PS D:\>
PS D:\>
PS D:\> $remhost
Id Name ComputerName ComputerType State ConfigurationName Available
-- -- -- -- -- -- -- -- -- --
3 Session3 10.10.10.100 RemoteMachine Opened Microsoft.PowerShell Avai
PS D:\> copy-item d:\copy.log -destination "c:\users\administrator\documents" -tosession $remhost
PS D:\>
```

Note in the after image, that the file copy.log is present in the Documents directory of the remote host.



The example below shows how a script may be executed on the remote host using a remote session. The script enumerates file system drives on a series of remote hosts. The script operates in this manner: a remote session is established for the current hosts; the files on the remote host are enumerated using the Invoke-Command cmdlet to execute a script block on the remote host; once the script block completes execution the session is closed.

```
# Script to retrieve disk capacity on remote hosts
clear-host
write-host `n`n
$cred = get-credential
$Servers = '10.10.10.100' # we may use an array of IP addresses
foreach ($server in $Servers) {

    # we assume the same credentials on each server
    $session = new-PSSession -ComputerName $server -Credential $cred

    # the script block executes on the remote host
    Invoke-Command -Session $session -ScriptBlock {

        $vols = (get-psdrive -PSProvider FileSystem).name # get an array
        of volume names
        $vols = $vols | foreach-object { $_ + ':' } # need to append ":" to the volume name

        write-host "`nDisk space for $env:computername"
        "{0,-6} {1,6} {2,6}" -f 'Vol','Size(GB)', 'Free(GB)', 'Used(GB)'

        foreach ($vol in $vols) {
            $d = [wmi] "Win32_LogicalDisk='$vol'"
            $usedspace = ($d.size - $d.Freespace)/1GB
            if ($d.size -ne $null ) { # only interested in drives that are mounted
                "{0,-6} {1,6} {2,6}" -f $vol,
                [math]::round($d.Size/1gb,1),
                [math]::round($d.FreeSpace/1gb,1),
                [math]::round($usedspace,1)
            }
        }
    }

    Remove-PSSession $session
}
```

```

1 # Script to retrieve disk capacity on remote hosts
2 clear-host
3 write-host `n`n
4 $cred = get-credential
5 $Servers = '10.10.10.100' # we may use an array of IP addresses
6 foreach ($server in $Servers) {
7
8     # we assume the same credentials on each server
9     $session = new-PSSession -ComputerName $server -Credential $cred
10
11    # the script block executes on the remote host
12    Invoke-Command -Session $session -ScriptBlock {
13
14        $vols = (get-psdrive -PSProvider FileSystem).name # get an array of volume names
15        $vols = $vols | foreach-object { $_ + ':' } # need to append ":" to the volume name
16
17        write-host "`nDisk space for $env:computername"
18        "{0,-6} {1,6} {2,6}" -f 'Vol', 'Size(GB)', 'Free(GB)', 'Used(GB)'
19
20        foreach ($vol in $vols) {
21            $d = [wmi] "Win32_LogicalDisk='$vol'"
22            $usedspace = ($d.size - $d.Freespace)/1GB
23            if ($d.size -ne $null) { # only interested in drives that are mounted
24                "{0,-6} {1,6} {2,6}" -f $vol,
25                [math]::round($d.Size/1gb,1),
26                [math]::round($d.FreeSpace/1gb,1),
27                [math]::round($usedspace,1)
28            }
29        }
30    }
31
32    Remove-PSSession $session
33 }

```

<

```

cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:

Disk space for CIS161-100
Vol      Size(GB) Free(GB)
C:       126.7   117.2

PS D:\>

```

In the above script, the cmdlet Remove-PSSession is used to close the session with the remote host. This is done in each iteration of the outer foreach loop to minimize resources at the local host. Each New-PSSession consumes local resources so an array of many hosts would consume a significant amount of resources locally as the script runs. Also, the session is no longer required once the drives of the remote host have been enumerated. So, using Remove-PSSession is an efficient way to write this script.

A script resident on the admin's host may be execute on the remote host using the FilePath parameter as below. Two syntax forms are provided in the example script below. The first does not use a remote session while the second uses the existing remote session specified by the variable \$remhost.

```
Invoke-Command -computername 10.10.10.100 -FilePath .\get-  
remoteprocess.ps1  
-credential $cred  
  
Invoke-Command -session $remhost -FilePath .\get-remoteprocess.ps1
```

```
PS C:\Users>  
PS C:\Users> get-content .\get-remoteprocess.ps1  
get-process | where-object { $_.name -like 'svc*' }  
PS C:\Users>  
PS C:\Users> Invoke-Command -ComputerName 10.10.10.100 -FilePath .\get-remoteprocess.ps1 -Credential  
Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName PSCom  
--- --- --- --- --- --- --- ---  
236 15 2612 4748 45 125.56 52 svchost 10.10  
674 50 12336 18104 1390 24.50 312 svchost 10.10  
442 29 13872 17276 902 20.80 592 svchost 10.10  
509 24 7244 13996 128 21.59 712 svchost 10.10  
382 13 3108 7740 36 42.72 720 svchost 10.10  
367 19 3680 5720 28 127.81 764 svchost 10.10  
359 33 10944 12108 60 1.34 816 svchost 10.10  
492 20 16084 17392 67 7,211.27 836 svchost 10.10  
39418 59 482108 483000 599 4,873.58 916 svchost 10.10  
474 25 5192 10484 87 92.17 952 svchost 10.10  
154 10 2908 8308 56 2.89 1392 svchost 10.10
```

The cmdlet Get-PSSession may be used to list the active sessions.

```
PS C:\Users>  
PS C:\Users> get-pssession  
Id Name ComputerName State ConfigurationName Availability  
-- -- -- -- -- --  
2 Session2 10.10.10.100 Opened Microsoft.PowerShell Available
```

Below is an interesting script that modifies the registry of a remote host. While Windows has many safeguards to protect against hacking exploits, there really is nothing that can be done to protect against a trusted insider with intent to do mischief as shown below.

The intent of the script below is to create a batch file on the remote host that, in this case, does something innocuous (with the implication that this batch file could do something destructive). The script then modifies the registry on the remote host such that the batch file is executed every time the remote host is restarted.

```
Enter-PSSession 10.10.10.100

# build a string that holds the fully qualified file name
$loc = Get-Location
$fn = $loc.ToString() + '\hello.bat'

# write the batch file to the current location
"echo hello world`npause" | Out-File $fn -Encoding ASCII

# Create a new string property for the Run key that contains the path
# to the batch file
New-ItemProperty -path
HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run -Name hello -
value "$fn"

# Exit the remote session
exit
```

The WMI Model

Sunday, April 07, 2013
1:45 PM

Windows Management Instrumentation (WMI) is a system of organized classes that represent management information about the Windows OS and other Windows-based hardware and software products. The classes are based upon the [Common Information Model](#) (CIM) as specified by the [Distributed Management Task Force](#). In WMI, the classes represent all things that WMI can manage. Each class has the properties and methods that provide the information and behaviors necessary to manage the underlying object.

Starting with PowerShell 3.0 CIM (Common Information Model) cmdlets are implemented providing a *superset* of the functionality of WMI. *CIM will eventually replace WMI.*

Each new version of Windows introduces improvements to WMI, including new WMI classes, as well as new capabilities for existing WMI classes. In many Microsoft products support for WMI continues to grow and expand. Some of the tasks you can perform with WMI follow:

- Report on drive configuration for locally attached drives, and for mapped drives.
- Report on available memory, both physical and virtual.
- Back up the event log.
- Modify the registry.
- Schedule tasks
- Share folders
- Switch from a static to a dynamic IP address
- Enable or disable a network adapter
- Defragment a hard disk drive

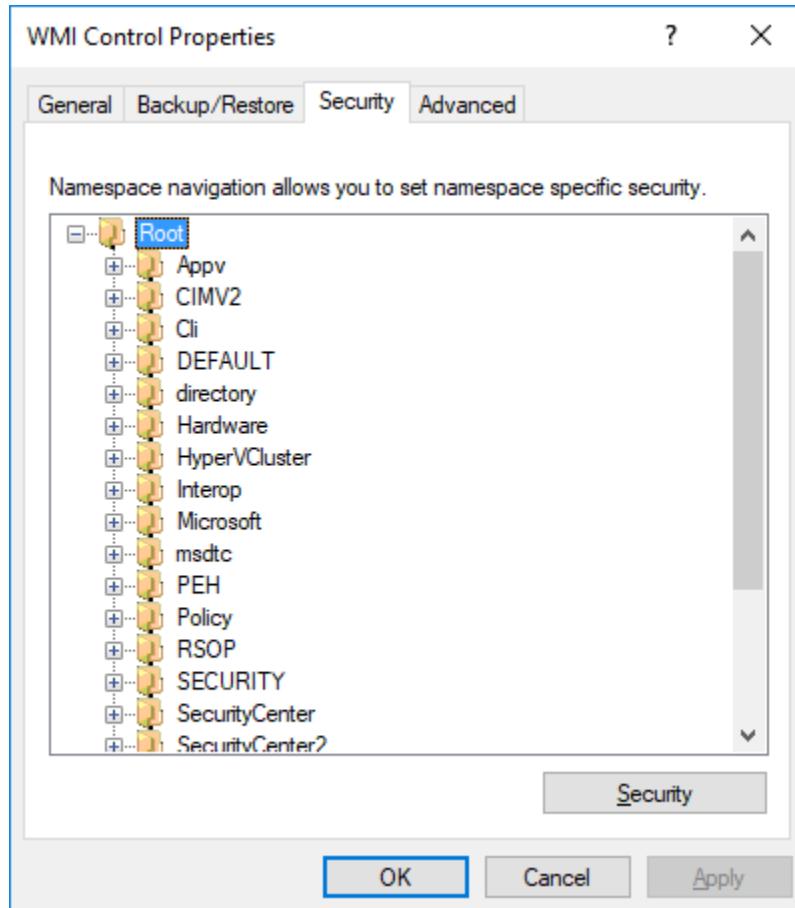
WMI uses a client-server architecture. The server in this case is the WMI service that must be running which may be accessed locally or remotely. The WMI service creates the object instances that are passed to the client, in this case PowerShell.

WMI organizes its classes in a hierarchy of *namespaces*. [Namespaces](#) are a way for grouping classes according to functionality. Since WMI supports thousands of classes, namespaces simplifies finding and using classes. For example, the namespace containing the core Windows OS classes is called **root\cimv2**, (the v2 indicates CIM version 2) while Microsoft IIS 6.0 stores its classes in **root\MicrosoftIISv2**.

WMI consists of three components: namespaces, providers and classes.

Namespaces:

The image below is accessible from Computer ⇒ Manage ⇒ Services and Applications ⇒ WMI Control ⇒ More Actions ⇒ Properties



The root\cimv2 namespace is the WMI default namespace.

To list the namespaces in the root enter the command below. Note the double underscore prefixing namespace.

```
get-wmiobject -namespace root -class __namespace | format-table name
```

```
PS>
PS> get-wmiobject -namespace root -class __namespace | format-wide name

subscription                               DEFAULT
CIMV2                                     msdtc
CLi                                         SECURITY
HyperVCluster                             SecurityCenter2
RSOP                                        PEH
StandardCimv2                            WMI
directory                                  Policy
virtualization                           Interop
Hardware                                    ServiceModel
SecurityCenter                           Microsoft
Appv
```

To list the namespaces in CIMV2 enter the command below:

```
Get-WmiObject -Class __Namespace -Namespace root/cimv2 | ft name
```

```
PS>
PS> Get-WmiObject -Class __Namespace -Namespace root/cimv2 | ft name

name
----
mdm
Security
power
ms_409
TerminalServices
Applications
```

Each of these namespaces have classes. The classes of interest are prefixed by `win32_`. These classes are found in `root/cimv2` and may be listed by the pipeline below.

```
get-wmiobject -namespace root\cimv2 -list |
? { $_.classpath -like '*Win32*' } |
sort name | format-table name
```

```
PS> get-wmiobject -namespace root\cimv2 -list | ? { $_.classpath -like '*Win32*' } | sort name | format-table name
```

Name

__Win32Provider
Win32_1394Controller
Win32_1394ControllerDevice
Win32_Account
Win32_AccountSID
Win32_ACE
Win32_ActionCheck
Win32_ActiveRoute
Win32_AllocatedResource
Win32_ApplicationCommandLine
Win32_ApplicationService
Win32_AssociatedProcessorMemory
Win32_AutochkSetting
Win32_BaseBoard
Win32_BaseService
Win32_Battery
Win32_Binary
Win32_BindImageAction
Win32_BIOS
Win32_BootConfiguration

The pipeline in the illustration below shows the number of "Win32" class available to provide information.

```
write-host `n`$($(( get-wmiobject -namespace root\cimv2 -list | ? { $_.classpath -like '*Win32*' } | measure-object ).count))`n' Win32 classes in WMI'
```

```
PS> write-host `n`>> $($(( get-wmiobject -namespace root\cimv2 -list | ? { $_.classpath -like '*Win32*' } | measure-object ).count))`n`>> 'Win32 classes in WMI'`n`>> 827 Win32 classes in WMI`n`PS>
```

Providers:

The WMI providers make the data associated with the classes available. The available providers is dependent upon the configuration of the Operating System. For example, for Windows Server the available providers is determined by roles and features installed for the server.

The available providers are may be listed by viewing the instances of the __provider class for the namespace.

```
get-wmiobject -class __provider -namespace root/cimv2 | sort name | ft name
```

```
PS>
PS> get-wmiobject -class __provider -namespace root/cimv2 | sort name | ft name

name
----
CIMWin32
Cimwin32A
DFSProvider
DskQuotaProvider
InvProv
MS_NT_EVENTLOG_EVENT_PROVIDER
MS_NT_EVENTLOG_PROVIDER
MS_Power_Management_Event_Provider
MS_Shutdown_Event_Provider
Msft_ProviderSubSystem
MSIProv
MSVDS__PROVIDER
MSVSS__PROVIDER
NamedJobObjectActgInfoProv
NamedJobObjectLimitSettingProv
NamedJobObjectProv
NamedJobObjectSecLimitSettingProv
ProviderSubSystem
```

Using WMI Classes

Monday, April 08, 2013
10:20 AM

WMI has three types of classes *core*, *common*, and *dynamic*. Core and common classes are templates used by developers of management applications. The dynamic classes provide actual information. Generally speaking the dynamic classes that provide data are prefixed by Win32_.

Because WMI consists of thousands of classes, finding the WMI class required may be problematic. While there is no easy method for find a specific class, we can filter classes by class name using wildcards. For example, to find classes related to tcp, the following may be entered:

```
get-wmiobject -list -class '*tcp*' | ? { $_.name -match '^Win32_' }
```

NameSpace: ROOT\cimv2		
Name	Methods	Properties
Win32_TCPIPPrinterPort	{}	{ByteCount, Caption, CreationClassName, Description, Frequency_Object}
Win32_PerfFormattedData_Tcpip_ICMP	{}	{Caption, Description, Frequency_Object}
Win32_PerfRawData_Tcpip_ICMP	{}	{Caption, Description, Frequency_Object}
Win32_PerfFormattedData_Tcpip_IC...	{}	{Caption, Description, Frequency_Object}
Win32_PerfRawData_Tcpip_ICMPv6	{}	{Caption, Description, Frequency_Object}
Win32_PerfFormattedData_Tcpip_IPv4	{}	{Caption, DatagramsForwardedPerSec, Description}
Win32_PerfRawData_Tcpip_IPv4	{}	{Caption, DatagramsForwardedPerSec, Description}
Win32_PerfFormattedData_Tcpip_IPv6	{}	{Caption, DatagramsForwardedPerSec, Description}
Win32_PerfRawData_Tcpip_IPv6	{}	{Caption, DatagramsForwardedPerSec, Description}
Win32_PerfFormattedData_Tcpip_NB...	{}	{BytesReceivedPerSec, BytesSentPerSec}
Win32_PerfRawData_Tcpip_NBTConne...	{}	{BytesReceivedPerSec, BytesSentPerSec}
Win32_PerfFormattedData_Tcpip_Ne...	{}	{BytesReceivedPerSec, BytesSentPerSec}
Win32_PerfRawData_Tcpip_NetworkA...	{}	{BytesReceivedPerSec, BytesSentPerSec}
Win32_PerfFormattedData_Tcpip_Ne...	{}	{BytesReceivedPerSec, BytesSentPerSec}
Win32_PerfRawData_Tcpip_NetworkI...	{}	{BytesReceivedPerSec, BytesSentPerSec}
Win32_PerfFormattedData_Tcpip_TCPV4	{}	{Caption, ConnectionFailures, ConnectionFailureRate}
Win32_PerfRawData_Tcpip_TCPV4	{}	{Caption, ConnectionFailures, ConnectionFailureRate}
Win32_PerfFormattedData_Tcpip_TCPV6	{}	{Caption, ConnectionFailures, ConnectionFailureRate}
Win32_PerfRawData_Tcpip_TCPV6	{}	{Caption, ConnectionFailures, ConnectionFailureRate}
Win32_PerfFormattedData_Tcpip_UDPv4	{}	{Caption, DatagramsNoPortPerSec, Description}
Win32_PerfRawData_Tcpip_UDPv4	{}	{Caption, DatagramsNoPortPerSec, Description}
Win32_PerfFormattedData_Tcpip_UDPv6	{}	{Caption, DatagramsNoPortPerSec, Description}
Win32_PerfRawData_Tcpip_UDPv6	{}	{Caption, DatagramsNoPortPerSec, Description}
Win32_PerfFormattedData_TCPIPCou...	{}	{Caption, DeniedConnectorsSendRequestRate}
Win32_PerfRawData_TCPIPCounters...	{}	{Caption, DeniedConnectorsSendRequestRate}
Win32_PerfFormattedData_WinNatCo...	{}	{Caption, Description, Frequency_Object}
Win32_PerfRawData_WinNatCounters...	{}	{Caption, Description, Frequency_Object}

The Win32_SystemBIOS class:

WMI provides BIOS information through the *BIOS* classes. Similarly find classes related to the BIOS of a computer

```
get-wmiobject -list '*bios*' | ? { $_.name -match '^Win32_' }
```

```
PS C:\Users>
PS C:\Users> get-wmiobject -list '*bios*' | ? { $_.name -match '^Win32_' }

NameSpace: ROOT\cimv2

Name                                Methods          Properties
----                                -----          -----
Win32_BIOS                           {}              {BiosCharacteristics, BIOSVersion}
Win32_SMBIOSMemory                   {SetPowerState, R... {Access, AdditionalErrorData}
Win32_SystemBIOS                     {}              {GroupComponent, PartComponent}
```

To query the BIOS information about the computer we would query the Win32_SystemBIOS class:

```
get-wmiobject -class win32_systembios
```

```
PS C:\Users>
PS C:\Users> get-wmiobject -class win32_systembios

__GENUS          : 2
__CLASS         : Win32_SystemBIOS
__SUPERCLASS    : CIM_SystemComponent
__DYNASTY       : CIM_Component
__RELPATH        : Win32_SystemBIOS.GroupComponent="\\\\\\EL-CID\\root\\\\cimv2:Win32_SystemBIOS.Name=\\4801\\",SoftwareElementID=0,Version=\\ALASKA - 1072009\\"
__PROPERTY_COUNT: 2
__DERIVATION    : {CIM_SystemComponent, CIM_Component}
__SERVER        : EL-CID
__NAMESPACE     : root\cimv2
__PATH          : \\\EL-CID\\root\\cimv2:Win32_SystemBIOS.GroupComponent="\\\\\\EL-CID\\root\\\\cimv2:Win32_BIOS.Name=\\4801\\",SoftwareElementID=0,Version=\\ALASKA - 1072009\\"
GroupComponent  : \\\EL-CID\\root\\cimv2:Win32_ComputerSystem.Name=\\EL-CID\\
PartComponent   : \\\EL-CID\\root\\cimv2:Win32_BIOS.Name=\\4801\\",SoftwareElementID=4801,Version=\\ALASKA - 1072009\\"
PSComputerName  : EL-CID
```

The property of interest is PartComponent that provides information about the BIOS. We create a subexpression to isolate the object then reference the property.

```
(get-wmiobject -class win32_systembios).partcomponent
```

```
PS C:\Users> PS C:\Users> (get-wmiobject -class win32_systembios).partcomponent  
\\"EL-CID\root\cimv2:Win32_BIOS.Name="4801",SoftwareElementID="4801",SoftwareElementState=3  
A - 1072009"
```

We would like to format this displays so that the information is more readable. The above cmdlet syntax isolates the partcomponent property which is a NoteProperty type. Since NoteProperty is a string type we may use the split method to produce an array of strings split on the comma.

```
(get-wmiobject -class win32_systembios).partcomponent.split(',')
```

```
PS C:\Users> PS C:\Users> ((get-wmiobject -class win32_systembios).partcomponent).split(',')  
\\"EL-CID\root\cimv2:Win32_BIOS.Name="4801"  
SoftwareElementID="4801"  
SoftwareElementState=3  
TargetOperatingSystem=0  
Version="ALASKA - 1072009"
```

The Win32_BIOS class:

The Win32_BIOS class is a superset of the Win32_SystemBIOS class providing information in a much more useable form. This class includes mostly properties by also a few methods that allow the administrator to rename the system and join the computer a domain or workgroup.

```
get-wmiobject -class win32_bios | format-list *
```

```

PS>
PS> get-wmiobject -class win32_bios | format-list *

PSComputerName          : EL-CID
Status                  : OK
Name                    : 4701
Caption                : 4701
SMBIOSPresent           : True
__GENUS                 : 2
__CLASS                 : Win32_BIOS
__SUPERCLASS             : CIM_BIOSElement
__DYNASTY                : CIM_ManagedSystemElement
__RELPATH                : Win32_BIOS.Name="4701",SoftwareElement
                           System=0,Version="ALASKA - 1072009"
__PROPERTY_COUNT          : 31
__DERIVATION              : {CIM_BIOSElement, CIM_SoftwareElemen
__SERVER                 : EL-CID
__NAMESPACE               : root\cimv2
__PATH                   : \\EL-CID\root\cimv2:Win32_BIOS.Name=
                           te=3,TargetOperatingSystem=0,Version=
                           {7, 10, 11, 12...}
                           {ALASKA - 1072009, 4701, American Me
BiosCharacteristics      :
BIOSVersion              : 4701
BuildNumber              : 255
CodeSet                  :
CurrentLanguage           : en|US|iso8859-1
Description              : 4701
EmbeddedControllerMajorVersion : 255
EmbeddedControllerMinorVersion : 255
IdentificationCode        :
InstallableLanguages      : 9
InstallDate              :
LanguageEdition           :
ListOfLanguages           : {en|US|iso8859-1, fr|FR|iso8859-1, es|ES|iso8859-1}
Manufacturer              : American Megatrends Inc.
OtherTargetOS             :
PrimaryBIOS               : True
ReleaseDate              : 20140506000000.000000+000
SerialNumber              : System Serial Number
SMBIOSBIOSVersion         : 4701
SMBIOSMajorVersion         : 2
SMBIOSMinorVersion         : 7
SoftwareElementID          : 4701
SoftwareElementState       : 3
SystemBiosMajorVersion     : 4
SystemBiosMinorVersion     : 6
TargetOperatingSystem      : 0
Version                  : ALASKA - 1072009

```

The Win32_ComputerSystem class:

The Win32_ComputerSystem class provides specific information about the computer. This class includes mostly properties by also a few methods that allow the administrator to rename the system and join the computer a domain or workgroup.

All properties and methods for Win32_ComputerSystem may be listed by entering following:

```
get-wmiobject -class Win32_ComputerSystem | format-list *
```

```
PS C:\Users>
PS C:\Users> get-wmiobject -class Win32_ComputerSystem | format-list *

PSCo mputerName : IS017871
AdminPasswordStatus : 3
BootupState : Normal boot
ChassisBootupState : 3
KeyboardPasswordStatus : 3
PowerOnPasswordStatus : 3
PowerSupplyState : 3
PowerState : 0
FrontPanelResetStatus : 3
ThermalState : 3
Status : OK
Name : IS017871
PowerManagementCapabilities :
PowerManagementSupported :
__GENUS :
__CLASS : Win32_ComputerSystem
__SUPERCLASS : CIM_UnitaryComputerSystem
__DYNASTY : CIM_ManagedSystemElement
__RELPATH : Win32_ComputerSystem.Name="IS017871"
```

The namespace parameter is omitted in the above command since root\cimv2 is the default namespace and may be omitted.

Since we are not interested in the internal classes (prefixed by __) we may exclude them as shown below:

```
gwmi Win32_ComputerSystem | select-object -property * -  
excludeproperty '__*' | fl *
```

```
PS C:\Users>  
PS C:\Users> gwmi Win32_ComputerSystem | select-object -property * -excludeproperty '__*' |  
  
PSComputerName : IS017871  
AdminPasswordStatus : 3  
BootupState : Normal boot  
ChassisBootupState : 3  
KeyboardPasswordStatus : 3  
PowerOnPasswordStatus : 3  
PowerSupplyState : 3  
PowerState : 0  
FrontPanelResetStatus : 3  
ThermalState : 3  
Status : OK  
Name : IS017871  
PowerManagementCapabilities :  
PowerManagementSupported :  
AutomaticManagedPagefile : True  
AutomaticResetBootOption : True  
AutomaticResetCapability : True  
BootOptionOnLimit :  
BootOptionOnWatchDog :  
BootROMSupported : True  
BootStatus : {0, 0, 0, 0...}  
Caption : IS017871
```

For this class, gwmi returns only one object for the computer on which it is running. Properties and methods for this may be accessed using the following syntax:

```
$comp = gwmi Win32_ComputerSystem | select-object -property * -  
excludeproperty '__'*  
$comp.BootupState  
$comp.PSComputerName
```

```
PS C:\Users>  
PS C:\Users> $comp = gwmi Win32_ComputerSystem | select-object -property * -excludeproperty '__'*  
PS C:\Users> $comp.BootupState  
Normal boot  
PS C:\Users> $comp.PSComputerName  
IS017871  
PS C:\Users>
```

Alternatively, we may use the syntax below omitting the need for a variable to hold the object:

```
(gwmi Win32_ComputerSystem ).BootupState  
(gwmi Win32_ComputerSystem ).PSComputerName
```

```
PS C:\Users>
PS C:\Users> (gwmi Win32_ComputerSystem).BootupState
Normal boot
PS C:\Users> (gwmi Win32_ComputerSystem).PSComputerName
IS017871
```

To find information about another computer, the -computername parameter may be used as below:

```
gwmi Win32_ComputerSystem -computername 10.10.10.100
```

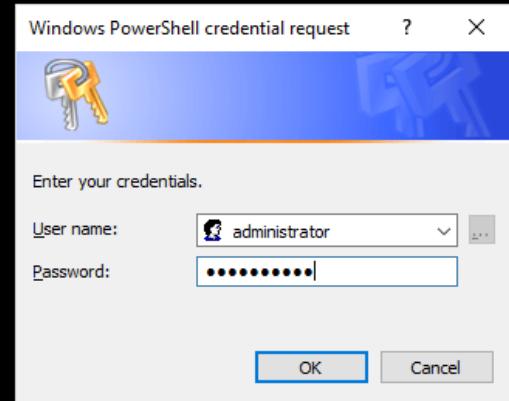
```
PS C:\Users>
PS C:\Users> gwmi Win32_ComputerSystem -computername 10.10.10.100

Domain          : CIS161-100.local
Manufacturer    : Microsoft Corporation
Model           : Virtual Machine
Name            : CIS161-100
PrimaryOwnerName : Windows User
TotalPhysicalMemory : 2147012608
```

When performing a WMI query against a remote computer, the user account performing the connection must be a member of the local administrators group on the remote machine or a domain administrator if the remote computer is joined to a domain. The remote computer was also be configured for remote management. If the common credentials are not present or the remote host is not a member of the domain, credentials must be provided as shown below.

```
gwmi Win32_ComputerSystem -computername 10.10.10.100 -Credential
(Get-Credential)
```

```
PS C:\Users>
PS C:\Users> gwmi Win32_ComputerSystem -computername 10.10.10.100 -Credential $(Get-Credential)
cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
```



```
PS C:\Users>
PS C:\Users> gwmi Win32_ComputerSystem -computername 10.10.10.100 -Credential $(Get-Credential)
cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential

Domain          : CIS161-100.local
Manufacturer    : Microsoft Corporation
Model           : Virtual Machine
Name            : CIS161-100
PrimaryOwnerName : Windows User
TotalPhysicalMemory : 2147012608
```

The Win32_LogicalDisk class:

The Win32_logicalDisk class returns information about all logical disks, i.e., local disks, network shares, optical media, etc.

```
gwmi win32_logicaldisk
```

```
PS C:\Users>
PS C:\Users> gwmi win32_logicaldisk

DeviceID      : C:
DriveType     : 3
ProviderName  :
FreeSpace     : 21804900352
Size          : 319231619072
VolumeName    :

DeviceID      : D:
DriveType     : 5
ProviderName  :
FreeSpace     :
Size          :
VolumeName    :
```

A filter may be applied to retrieve specific disk types. For example, to retrieve only local disks do the following:

```
gwmi win32_logicaldisk -filter drivetype=3
```

```
PS C:\Users>
PS C:\Users> gwmi win32_logicaldisk -filter drivetype=3

DeviceID      : C:
DriveType     : 3
ProviderName  :
FreeSpace     : 21804019712
Size          : 319231619072
VolumeName    :
```

Information about the `win32_logicaldisk` class including the description of the `drivetype` property is available at [http://msdn.microsoft.com/en-us/library/aa394173\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/aa394173(v=vs.85).aspx)

To determine the amount of free space for local drives:

```
gwmi win32_logicaldisk -filter drivetype=3 | measure-object -  
property freespace  
-maximum -minimum
```

```
PS C:\Users>  
PS C:\Users> gwmi win32_logicaldisk -filter drivetype=3 | measure-object -property freespace -maximum -minimum  
  
Count    : 1  
Average   :  
Sum      :  
Maximum   : 21803950080  
Minimum   : 21803950080  
Property  : freespace
```

WMI Query Language

Monday, June 5, 2017
1:52 PM

WQL is a SQL-like syntax for querying WMI. WQL is useful when WMI queries must be built dynamically in a script.

Basic WQL syntax requires a *select* phrase to select the properties of interest, a *from* phrase to indicate the class to use, and a *where* phrase to specify criteria for the query.

```
# Query for service objects whose DisplayName begins with "Windows"  
$qry = 'select * from Win32_Service where displayname like  
"Windows%"'  
get-wmiobject -query $qry | format-list *
```



```
# Query for service objects whose DisplayName begins with "Windows"  
$qry = 'select displayName, state from Win32_Service where  
displayname like "Windows%"'  
get-wmiobject -query $qry
```

Additional predicates may be added to the where clause:

```
# same as the above query but only return objects for running services
$qry = 'select DisplayName
        from Win32_Service
        where displayname like "Windows%" and state = "Running"'
Get-wmiobject -query $qry
```

Note in the above queries, the complete query is delimited by single quotes. This is important since literals in the query must be delimited by double quotes.

A tutorial for WQL is available at

<http://www.darkoperator.com/blog/2013/3/11/introduction-to-wmi-basics-with-powershell-part-3-wql-and-se.html>

Accelerators

Monday, April 08, 2013
10:35 AM

Type accelerators are available to reduce the complexity of the syntax and make these types conform to the standard typing syntax of PowerShell. Accelerators are available for many interfaces including WMI.

PowerShell provides the type accelerator [wmi].

The traditional method for accessing an WMI object is shown below. The example below retrieves the Windows Search service object.

```
$s = get-wmiobject -name Win32_Service -filter ' Name = "WSearch" '
```

Using the type accelerator, the syntax above may be reduced to the following:

```
$s = [wmi] "Win32_Service.Name='Wsearch'"
```

Spacing is very important in the above syntax, extraneous spaces in the string following the type accelerator will throw an error.

The following script use the [wmi] type accelerator to list information about the volumes in the local computer.

```
clear-host
write-host
$vols = 'c:', 'd:'
"{0,3} {1,10} {2,10} {3,10}" -f 'Vol', 'Size', 'Free', 'Used'
foreach ($vol in $vols) {
    $d = [wmi] "Win32_LogicalDisk=$vol"
    $usedspace = ($d.size - $d.Freespace)/1GB

    $size = [math]::round($d.Size/1gb,1)
    $free = [math]::round($d.FreeSpace/1gb,1)
    $used = [math]::round($usedspace,1)
    "{0,3} {1,10} {2,10} {3,10}" -f $vol, $size, $free, $used
}
```

```
1  clear-host
2  write-host
3  $vols = 'c:', 'd:'
4  "{0,3} {1,10} {2,10} {3,10}" -f 'Vol', 'Size', 'Free', 'Used'
5  foreach ($vol in $vols) {
6      $d = [wmi] "Win32_LogicalDisk=$vol"
7      $usedspace = ($d.size - $d.Freespace)/1GB
8
9      $size = [math]::round($d.Size/1gb,1)
10     $free = [math]::round($d.FreeSpace/1gb,1)
11     $used = [math]::round($usedspace,1)
12     "{0,3} {1,10} {2,10} {3,10}" -f $vol, $size, $free, $used
13 }
14 write-host
```

Vol	Size	Free	Used
c:	460.5	404.6	55.9
d:	0	0	0

Secure Strings

Monday, April 15, 2013

4:42 PM

Using authentication credentials for use in scripts has always been problematic. Historically credentials, i.e., user id and password, read by scripts were stored as strings (system.string in .Net) and could be easily exposed. The problem arises in the plain strings are *immutable*, i.e., cannot be changed once created. If a string is changed, a new version of the string is created. Additionally once created, strings cannot be cleared from memory until the garbage collect process runs.

```
[string] $pwd = read-host -prompt 'Enter password'  
$pwd = $pwd.trim() # trim spaces
```

In the above, after the second statement executes, two copies of the string exist in memory: the original user entry and the trimmed version residing in \$pwd. The password strings remain in memory until the PowerShell *garbage collector* executes and removes unused objects from memory. The issue is that both strings containing the clear text password are resident in memory. Taking a memory dump before the garbage collector executes easily exposes string contents. This issue is further compounded since the password is stored in clear text.

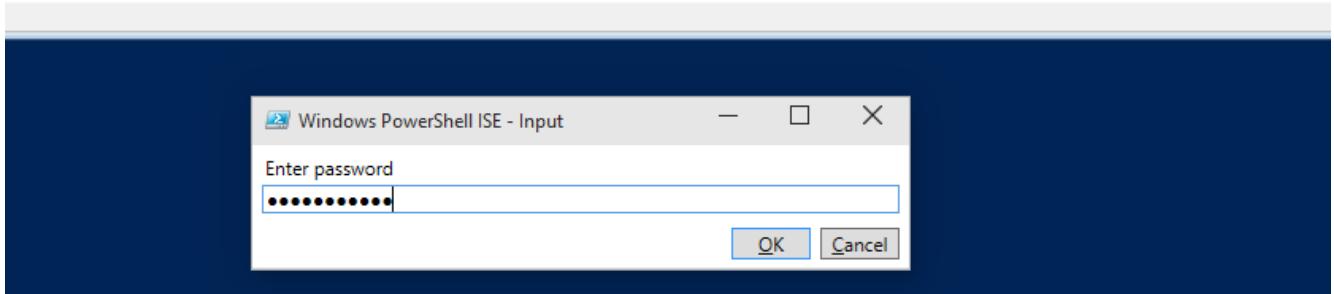
To solve this problem, .Net introduced the [System.SecureString] type. A *SecureString* is mutable, i.e., can be changed but never copied. This type also implements a dispose method which removes the string from memory. This provides the script developer the ability to use an encrypted password then disposing of it when it's no longer needed.

```
$encryptedPwd = read-host -asSecureString -prompt 'Enter password'  
write-host "This is what you get when you display a secure string  
$encryptedPwd "  
  
$marshal = [System.Runtime.InteropServices.Marshal]  
# Get a pointer to the secure string (remember it cannot be copied)  
$ptr = $marshal::SecureStringToBSTR( $encryptedPwd )
```

```
# Convert it to a string
$clearPwd = $marshal::PtrToStringBSTR( $ptr )
write-host "The clear text password is $clearPwd"

$marshal::ZeroFreeBSTR( $ptr ) # clear out the mem
```

```
1 clear-host
2
3 $encryptedPwd = read-host -asSecureString -prompt 'Enter password'
4 write-host "This is what you get when you display a secure string $encryptedPwd"
5
6 $marshal = [System.Runtime.InteropServices.Marshal]
7 # Get a pointer to the secure string (remember it cannot be copied)
8 $ptr = $marshal::SecureStringToBSTR( $encryptedPwd )
9 # Convert it to a string
10 $clearPwd = $marshal::PtrToStringBSTR( $ptr )
11 write-host "The clear text password is $clearPwd"
12
13 $marshal::ZeroFreeBSTR( $ptr ) # clear out the mem
```



After entering the password, the console shows the display of the secure string. Following that is the password that was unencrypted and displayed on the console.

```

1 clear-host
2
3 $encryptedPwd = read-host -asSecureString -prompt 'Enter password'
4 write-host "This is what you get when you display a secure string $encryptedPwd "
5
6 $marshal = [System.Runtime.InteropServices.Marshal]
7 # Get a pointer to the secure string (remember it cannot be copied)
8 $ptr = $marshal::SecureStringToBSTR( $encryptedPwd )
9 # Convert it to a string
10 $clearPwd = $marshal::PtrToStringBSTR( $ptr )
11 write-host "The clear text password is $clearPwd "
12
13 $marshal::ZeroFreeBSTR( $ptr ) # clear out the mem

```

```

<
This is what you get when you display a secure string System.Security.SecureString
The clear text password is cntpassword
PS C:\WINDOWS\System32> |

```

Credentials

Monday, April 15, 2013
5:42 PM

.Net also assists in securing collecting authentication credentials for use in scripts. The cmdlet *get-credentials* securely obtains credentials.

```

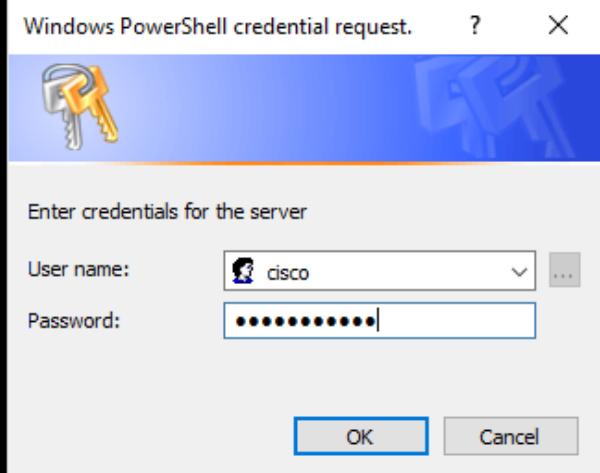
$cred = get-credential -message 'Enter credentials for the server'

write-host "Contents of `\$cred are $cred"
write-host "Contents of `\$cred.Username are $($cred.Username)"
write-host "Contents of `\$cred.Password are $($cred.Password)"

$ptr = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR(
$cred.Password )
$clrpwd = [System.Runtime.InteropServices.Marshal]::PtrToStringBSTR(
$ptr )
write-host "Clear text password is $clrpwd"

```

```
PS C:\Users>
PS C:\Users> $cred = get-credential -message 'Enter credentials for the server'
```

A screenshot of a Windows PowerShell credential request dialog box. The title bar says "Windows PowerShell credential request." It features a key icon on the left. The main area has a blue background with a faint "Windows" watermark. The text "Enter credentials for the server" is centered. There are two input fields: "User name:" with a dropdown arrow containing "cisco" and a "..." button, and "Password:" with a masked input field showing "*****". At the bottom are "OK" and "Cancel" buttons.

After the credentials are entered this console shows the following.

```
PS C:\Users>
PS C:\Users> $cred = get-credential -message 'Enter credentials for the server'
PS C:\Users>
PS C:\Users> write-host "Contents of `\$cred are $cred"
Contents of $cred are System.Management.Automation.PSCredential
PS C:\Users> write-host "Contents of `\$cred.Username are $($cred.Username)"
Contents of $cred.Username are cisco
PS C:\Users> write-host "Contents of `\$cred.Password are $($cred.Password)"
Contents of $cred.Password are System.Security.SecureString
PS C:\Users>
PS C:\Users> $ptr = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($cred.Password)
PS C:\Users> $clrpwd = [System.Runtime.InteropServices.Marshal]::PtrToStringBSTR($ptr)
PS C:\Users> write-host "Clear text password is $clrpwd"
Clear text password is cncpassword
```

The entered credentials may then be used in an cmdlet that is executing against a remote host.

One of the problems with the approach above is that scripts may be scheduled to run unattended. So, how should the credentials be protected. One approach is shown below.

The first step is to save the password of the admin account as a securestring in a text file. This text file may now be used in production scripts to establish credentials. In the script below, the cmdlet *convertfrom-secure* string converts the secure string into a standard encrypted string. Unlike a secure string, an standard encrypted string may be saved to a text file.

```
read-host 'Enter admin password' -assecurestring |  
convertfrom-securestring | out-file .\securestring.txt  
  
get-content .\securestring.txt
```

```
PS C:\Users>  
PS C:\Users> read-host 'Enter admin password' -assecurestring |  
>> convertfrom-securestring | out-file .\securestring.txt  
Enter admin password: *****  
PS C:\Users>  
PS C:\Users> get-content .\securestring.txt  
01000000d08c9ddf0115d1118c7a00c04fc297eb010000002125b865510fa346bd32  
0000fcc20b560f66c7abadd74a07b7bf39e44712bb2d3e67f54485c57ff042c37c6  
0bebfc586793e2546fa533536235bb01b3bfc3d4d19e200000006ee27678db0a9afc  
0000bd37fc12cd59a9eef73eb85488f1bc63166859bb2dbd5f98b6a75667058b6740  
fbc1d252073f
```

In the production script, read the encrypted password for the admin account and create a credentials object to use for authentication. The cmdlet *convertto-securestring* converts a standard encrypted string to a secure string.

```
$passwd = get-content .\securestring.txt | Convertto-SecureString  
$username = 'admin'  
$cred = new-object -typename  
System.Management.Automation.PSCredential `  
-argumentlist $username, $passwd  
write-host "The contents of `$cred are $cred"
```

```
PS C:\Users>  
PS C:\Users> $passwd = get-content .\securestring.txt | Convertto-SecureString  
PS C:\Users> $username = 'admin'  
PS C:\Users> $cred = new-object -typename System.Management.Automation.PSCredential  
>> -argumentlist $username, $passwd  
PS C:\Users> write-host "The contents of `$cred are $cred"  
The contents of $cred are System.Management.Automation.PSCredential  
PS C:\Users>
```

The variable \$cred may then be used in any cmdlet that access a remote host and needs to authenticate using those credentials.

If you are not too concerned about security (and you should be), you may embed the credentials as follows.

```
$passwd = ConvertTo-SecureString 'mypasswd' -asplaintext -force
```

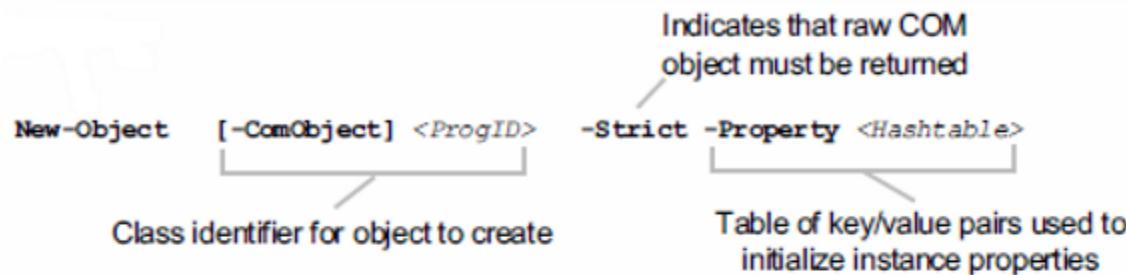
```
$username = 'myaccount'  
$cred = New-Object  
System.Management.Automation.PSCredential($username,$passwd)
```

Introduction

Tuesday, April 8, 2014
8:38 PM

Component Object Module (COM) is a Microsoft technology that allows libraries to be written that may be accessed from multiple languages or environments. The COM specification allows developers to create re-useable software components. COM allows running applications to expose automation interfaces that external programs could use to remotely control them. COM classes available in PowerShell provide easy access to many Windows features and applications.

The new-object cmdlet is used to create COM objects.



"COM objects are one of these adapted types, but the way the PowerShell adapter works is affected by the presence or absence of a COM Interop library. In effect, this Interop library is .NET's own adaptation layer for COM. The net effect is that the PowerShell COM adapter will project a different view of a COM object if an Interop library is loaded versus when there's no Interop library. This becomes a problem because, for any given COM class on any given machine, there may or may not be an Interop library, so you may or may not get the doubly adapted COM object. If you want to be able to write scripts that behave consistently everywhere, you need a way to control how the adaptation is done. The **-Strict** parameter allows you to detect this when an Interop library is loaded. Once you know what's happening, you can decide whether you want to fail or continue along a different code path."

"Windows PowerShell in Action", Bruce Payette

Some common COM classes that are available:

Shell.Application	Provides access to File Explorer and its capabilities. Allows automation of many shell tasks like opening file browser windows; launching documents or the help system; finding printers, computers, or files; and so on.
SAPI.SpVoice	Provides access to the Microsoft Speech API. Allows text-to-speech conversion with the <code>Speak("string")</code> method.
Schedule.Service	Allows you to create and schedule tasks, get a list of running tasks, and delete task definitions.
WScript.Shell	Provides access to system information and environment variables. Lets you create shortcuts and work with the Registry.

COM objects are registered in the Registry. The script below lists the registered objects on the computer where the script is run:

```
$classes = @()
foreach ($class in (gci "REGISTRY::HKey_Classes_Root\clsid\*\*\progid" )) {
    if ($class.name -match '\*\*\$') {
        $classes += $class.GetValue(""))
    }
}
$classes | sort | out-host -paging
```

System.Random

Sunday, April 13, 2014
9:05 AM

System.Random:

This COM object generates pseudorandom numbers. Three methods are available: Next which generates integers; NextDouble which generates double precision floating point numbers; and NextBytes which generates integers with a value between 1 and 255. Note, 255 is the largest number that can fit into a byte.

```
PS C:\Users>
PS C:\Users> $classes | select-string 'random'
DXImageTransform.Microsoft.RandomBars.1
System.Random
DXImageTransform.Microsoft.RandomDissolve.1
```

```
$rand = new-object -ComObject system.random

# generate a random integer
$r = $rand.Next()
```

```

write-host "The random number is $r"

# generate a random double
$r = $rand.NextDouble()
write-host "The random double precision floating point number is $r"

# generate a byte array
[byte[]] $b = 1..10
$rand.NextBytes($b)
$b

```

```

PS C:\Users>
PS C:\Users> $rand = new-object -ComObject system.random
PS C:\Users>
PS C:\Users> # generate a random integer
PS C:\Users> $r = $rand.Next()
PS C:\Users> write-host "The random number is $r"
The random number is 244805432
PS C:\Users>
PS C:\Users> # generate a random double
PS C:\Users> $r = $rand.NextDouble()
PS C:\Users> write-host "The random double precision floating point number is $r"
The random double precision floating point number is 0.137075572804117
PS C:\Users>
PS C:\Users> # generate a byte array
PS C:\Users> [byte[]] $b = 1..10
PS C:\Users> $rand.NextBytes($b)
PS C:\Users> $b
24
75
162
108
49
129
128
160
199
143
.

```

The `NextBytes` method is particularly useful for generating random passwords. The script below uses the `system.random` COM object to generate password strings.

```

function getRandomChars {
    <#
        this function generates an array of printable
        ASCII characters.  The characters are biased toward
        letters and numbers.
    #>

```

```

[char[]] $passchars = @()
<# do this three times to have more letters and numbers
   than special characters #>
for ($i=1; $i -le 3; $i++) {
    # add uppercase alphabet
    65..90 | % { $passchars += [char] $_ }
    # add lowercase alphabet
    97..126 | % { $passchars += [char] $_ }
    # add digits 0 through 9
    48..57 | % { $passchars += [char] $_ }
}
<# add the upper and lowercase letters,
   digits, special characters #>
33..126 | % { $passchars += [char] $_ }

<# This snippet scrambles the character array
   using the Fischer-Yates shuffle #>
$rand = new-object -ComObject system.random
for ($i=$passchars.length-1; $i -ge 0; $i--) {
    # get a random integer -le to the current loop index + 1
    $randindx = $rand.Next( $i + 1)
    # swap the characters
    $c = $passchars[$randindx]
    $passchars[$randindx] = $passchars[$i]
    $passchars[$i] = $c
}
return $passchars
}

clear-host
write-host "`t`tRandom Password Generator"
[char[]] $passchars = getRandomChars
$len = read-host "Password length"
[string] $passwd = ''
for ($i=0; $i -lt $len; $i++) {
    $passwd += $passchars[$i]
}
write-host "The password is $passwd"

```

Shell.Application

Tuesday, April 8, 2014
9:29 PM

This COM object provides access to the Windows File Explorer.

```
$wshell = new-object -com Shell.Application  
  
$wshell | gm  
  
# cascade the windows on the desktop  
$wshell.CascadeWindows()  
  
# start a new instance of File Explorer pointing to c:\  
$wshell.explore('c:\')  
  
# inspect the properties of the open File Explorer windows  
$wshell.Windows()
```

The *Windows* method above would normally return a collection object since many File Explorer windows could be open. So, under normal circumstances, we would use the indexes just like any other array.

```
($wshell.Windows())[0]  
  
$w = $wshell.Windows()  
$w[0]
```

The following example demonstrates the use of the *Windows* method.

```
$wshell = new-object -com Shell.Application  
  
# start a new instance of File Explorer pointing to c:\  
$wshell.explore('c:\')  
  
# Get object associated with File Explorer window  
# This would not work if more than one window is open  
$feo = $wshell.Windows()  
  
# hide the window  
$feo.Visible = $false  
$feo.Navigate('c:\users')
```

This script uses the shell.application COM object to zip a folder.

```
# folder containing files to zip
$FolderToZip = 'd:\test'
# the zip file folder
$ZipFile = $FolderToZip + '.zip'
# a zip file has the header consisting of the bytes below
$ziphdr = [byte[]]@( 80, 75, 5, 6 + (, 0 * 18 ) )
$ziphdr | Set-Content $ZipFile -encoding byte

$wshell = new-object -com shell.application
# the namespace method returns a folder object
$srcFolder = $wshell.NameSpace($FolderToZip)
$zipFolder = $wshell.NameSpace($ZipFile)
# items() represents all the files in the folder
$zipFolder.CopyHere($srcFolder.items())
```

Excel.Application

Tuesday, April 9, 2013
2:03 PM

The following is an interesting example of using PowerShell and the inter-process communication features provided by COM to create and populate an Excel worksheet.

```
# Start Excel and assign the process object to the variable $ExcelApp
$ExcelApp = New-Object -ComObject Excel.Application

# $ExcelApp | gm

$ExcelApp.Visible = $true # (optional) make the app visible

$Workbook = $ExcelApp.workbooks.add() # add a workbook
# $workbook | gm

# save the workbook
set-location "~\MyTemp" # set the location to MyTemp in the home directory
```

```

$wbname = "$(get-location)\mybook.xlsx"
$Workbook.SaveAs($wbname)

# $workbook.worksheets | gm
$WorkSheet = $Workbook.Worksheets.Item(1) # select the first sheet
$WorkSheet.name = 'mike'

# populate the cells in the first 10 rows in the first 10 columns
for ($i = 1; $i -le 10; $i++) {
    for ($j = 1; $j -le 10; $j++) {
        $WorkSheet.Cells.Item($i,$j) = $i + $j
    }
}

# add a SUM function to row 11 below each column
$col = 'A','B','C','D','E','F','G','H','I','J'
for ($j=1; $j -le 10; $j++) {
    $WorkSheet.Cells.Item(11,$j) = '=sum(' + $col[$j-1] + '1:' +
$col[$j-1] + '10)'
    $WorkSheet.Cells.Item(11,$j).Interior.ColorIndex = 27
    $WorkSheet.Cells.Item(11,$j).Font.Bold = $true
    $WorkSheet.Cells.Item(11,$j).Font.ColorIndex = 30
}

```

Other COM Objects

Wednesday, April 9, 2014
3:52 PM

InternetExplorer.Application:

This InternetExplorer.Application COM object provides access to the Internet Explorer application. It shares many of the properties and methods of the wshell.application object.

```

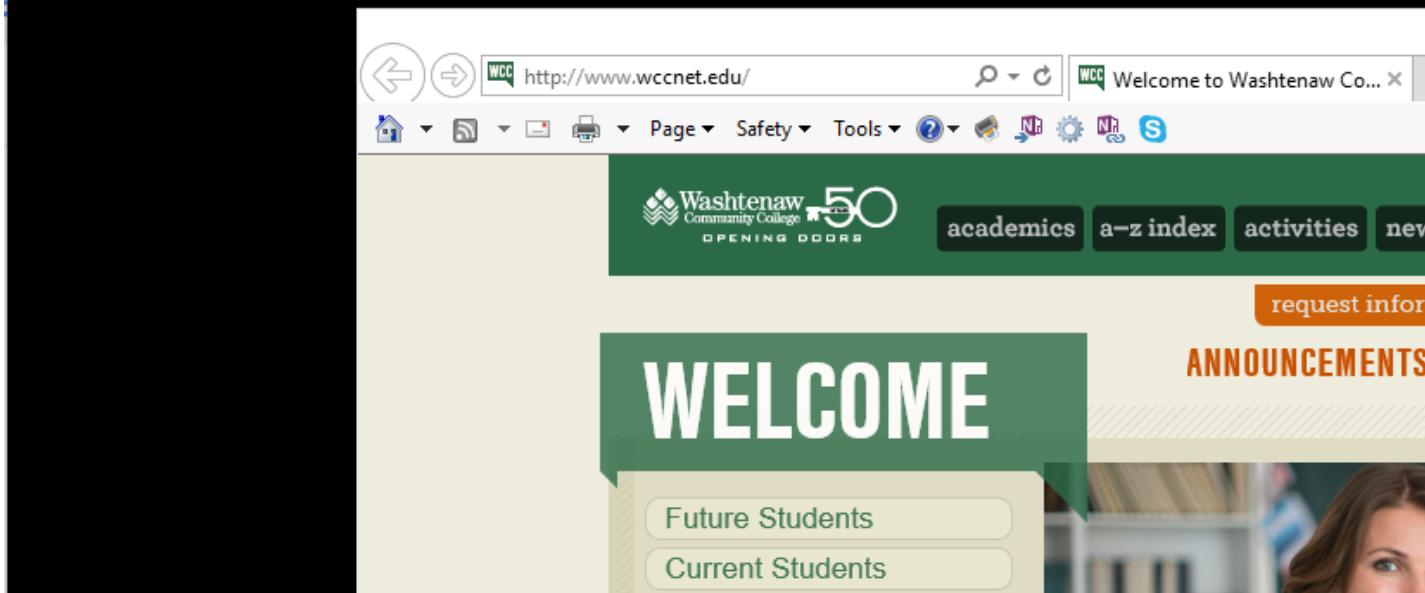
# instantiate the object
$ie = New-Object -ComObject InternetExplorer.Application

# navigate to a URL
$ie.Navigate('www.wccnet.edu')

```

```
# the application is running but invisible, make it visible  
$ie.Visible = $true
```

```
PS C:\Users> $ie = New-Object -ComObject InternetExplorer.Application  
PS C:\Users> $ie.Navigate('www.wccnet.edu')  
PS C:\Users> $ie.Visible = $true  
PS C:\Users>
```



This script is a basic web crawler that dumps the html of the home page using the INternetExplorer.Application COM object.

```
$ie = New-Object -ComObject InternetExplorer.Application  
  
foreach ($site in ('www.wccnet.edu', 'www.msnbc.com')) {  
    $ie.navigate($site)  
    start-sleep 1  
    $ie.document.body.innerhtml | out-file "$site.html"  
}  
$ie.quit
```

WinHttp.WinHttpRequest.5.1:

This object implements the http protocol. Another approach to creating a web crawler is to use the WinHttp.WinHttpRequest.5.1 COM object.

```

$http = New-Object -ComObject WinHttp.WinHttpRequest.5.1
foreach ($site in ('www.wccnet.edu', 'www.msnbc.com') ) {
    $url = 'http://' + $site
    $http.open('GET', $url, $false)
    $http.send()
    $html = $http.ResponseText
    $filename = $site.replace('.', '') + '.html'
    $html | out-file $filename -Encoding ascii
}

```

System.Security.Cryptography.HMACSHA1:

This COM object generates a Hashed Message Key Authentication code using the SHA1 hash algorithm. HMAC is used to ensure that messages send over an unsecure channel are not altered in transmission. HMAC is commonly used in VPNs and other secure protocols. The HMAC protocol requires a secret key known to both ends.

HMAC SHA2 "mixes a secret key with the message data, hashes the result with the hash function, mixes that hash value with the secret key again, and then applies the hash function a second time. The output hash is 160 bits in length" (Technet).

```

# generate an HMAC has for message
clear-host
write-host `n

# create a UTF8 encoding object
$enc = [system.Text.Encoding]::UTF8

# create the HMAC object
$sha1 = new-object -ComObject System.Security.Cryptography.HMACSHA1

# Convert the key and message strings to byte arrays
$sha1.key = $enc.GetBytes('cntpword')
$message = 'We attack at dawn. Pass it on'
[byte[]] $bytes = $enc.GetBytes($message)

# compute the hash as a byte array
$hashint = $sha1.ComputeHash($bytes)

[string] $hash = $null

```

```
# convert the hash to a hex string
$hash = -join $hashint.ForEach({$_.ToString('x2')})

write-host `"$message`" `t $hash"

write-host `n
```

One of the required properties of a good HMAC function is that the same message must always generate the same hash. Also, a small change in the message should result in a large change in the hash.

SAPI.Spvoice:

This COM object is used for text to speech applications. Below is a very simple implementation of this COM object.

```
$t = 'Friends, Romans, countrymen, lend me your ears; I come to bury
Caesar, not to praise him.
The evil that men do lives after them; The good is oft interred with
their bones;
So let it be with Caesar. The noble Brutus Hath told you Caesar was
ambitious:
If it were so, it was a grievous fault, And grievously hath Caesar
answered it.
Here, under leave of Brutus and the rest-- For Brutus is an
honourable man;
So are they all, all honourable men-- Come I to speak in Caesars
funeral.
He was my friend, faithful and just to me: But Brutus says he was
ambitious;
And Brutus is an honourable man. He hath brought many captives home
to Rome
Whose ransoms did the general coffers fill: Did this in Caesar seem
ambitious?
When that the poor have cried, Caesar hath wept: Ambition should be
made of sterner stuff:
Yet Brutus says he was ambitious; And Brutus is an honourable man.

$speak = new-object -com SAPI.Spvoice
$speak.speak($t)
```

Custom Objects

Monday, February 04, 2013
1:20 PM

Objects are structures that contain data and behaviors by which the data is altered.

To better illustrated objects in PS, let's look at services:

```
get-service | gm
```

TypeName:	object class
Name	MemberType
----	-----
Name	AliasProperty
RequiredServices	AliasProperty
Disposed	Event
Close	Method
Continue	Method
CreateObjRef	Method
Dispose	Method
Equals	Method
ExecuteCommand	Method
GetHashCode	Method
GetLifetimeService	Method
GetType	Method
InitializeLifetimeService	Method
Pause	Method
Refresh	Method
Start	Method
Stop	Method
WaitForStatus	Method
CanPauseAndContinue	Property
CanShutdown	Property
CanStop	Property
Container	Property
DependentServices	Property
DisplayName	Property
MachineName	Property
ServiceHandle	Property
ServiceName	Property
ServicesDependedOn	Property
ServiceType	Property
Site	Property
Status	Property
ToString	ScriptMethod

- Properties – Information about the object.
- Methods – Things we can do to the objects.
- ScriptMethods – Pieces of embedded code that will execute when called.
- Events – Things that happen to an object. A script may subscribe to the event to be notified when it happens.

A custom object may be created and assigned properties and methods just like any other PowerShell object. A custom object could also be created by taking an object intrinsic to PowerShell and extending it by adding properties and methods.

There are limitations as to the types of properties and methods that may be added. Consult the syntax diagrams for more information.

Creating Custom Objects

Friday, April 25, 2014
4:38 PM

A custom object is created using the *new-object* cmdlet with a type of *PSObject*. The syntax described as below:

```
<object variable> = New-Object -TypeName PSObject
```

Properties and methods are added to the custom object using the *add-member* cmdlet.

In this example we create a custom object called \$myhost which has two properties HostName and IPAddress. We first use the New-Object cmdlet to create an object devoid of any properties.

```
$myhost = New-Object -TypeName PSObject
```

Using the Add-Member cmdlet, we then add the properties HostName and IPAddress which are a NoteProperty type. NoteProperty types are just variable length string types.

```
Add-Member -inputobject $myhost -NotePropertyName HostName `  
-NotePropertyValue 'google-public-dns-a.google.com'  
Add-Member -inputobject $myhost -NotePropertyName IPAddress `  
-NotePropertyValue '8.8.8.8'
```

```
PS C:\Users>
PS C:\Users> $myhost = New-Object -TypeName PSObject
PS C:\Users>
PS C:\Users> Add-Member -inputobject $myhost -NotePropertyName HostName `>>> -NotePropertyValue 'google-public-dns-a.google.com'
PS C:\Users> Add-Member -inputobject $myhost -NotePropertyName IPAddress `>>> -NotePropertyValue '8.8.8.8'
PS C:\Users>
```

Inspecting this object with get-member, we see the results below.

```
PS C:\Users>
PS C:\Users> $myhost | get-member

    TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        -----      -----
Equals      Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType     Method      type GetType()
ToString    Method      string ToString()
HostName   NoteProperty string HostName=google-public-dns-a.google.com
IPAddress  NoteProperty string IPAddress=207.75.134.1
```

With the object created, it may be used as below. Note that the IPAddress property was changed but not the HostName, resulting in an inconsistency.

```
test-connection $myhost.IPAddress -count 1 | format-table
test-connection $myhost.HostName -count 1 | format-table
$myhost.IPAddress = '207.75.134.1'
test-connection $myhost.IPAddress -count 1 | format-table
```

```

PS C:\Users>
PS C:\Users> test-connection $myhost.IPAddress -count 1 | format-table
Source      Destination      IPV4Address      IPV6Address
----      ----      ----      -----
EL-CID      8.8.8.8      8.8.8.8

PS C:\Users> test-connection $myhost.HostName -count 1 | format-table
Source      Destination      IPV4Address      IPV6Address
----      ----      ----      -----
EL-CID      google-publi... 8.8.8.8

PS C:\Users> $myhost.IPAddress = '207.75.134.1'
PS C:\Users>
PS C:\Users> test-connection $myhost.IPAddress -count 1 | format-table
Source      Destination      IPV4Address      IPV6Address
----      ----      ----      -----
EL-CID      207.75.134.1

```

An alternative is to add the properties at the time of creation of the object is the -Property parameter with a hash table of properties.

```
$myhost = New-Object -TypeName PSObject -Property
@{HostName='google-public-dns-a.google.com'; IPAddress='8.8.8.8'}
```

For sake of readability, the hash table may be assigned to a variable and the variable used in the object creation:

```
$props = @{HostName='google-public-dns-a.google.com';
IPAddress='8.8.8.8'}
$myhost = New-Object -TypeName PSObject -Property $props
```

Another type of property that we may use for custom objects is the AliasProperty. This property is useful that we may use shorter names to reference the property.

```
Add-Member -inputobject $myhost -Membertype AliasProperty -name name
-value HostName
Add-Member -inputobject $myhost -Membertype AliasProperty -name ip -
-value IPAddress
```

```

PS C:\Users> Add-Member -inputobject $myhost -Membertype AliasProperty -name name -value HostName
PS C:\Users> Add-Member -inputobject $myhost -Membertype AliasProperty -name ip -value IPAddress
PS C:\Users> test-connection $myhost.ip -count 1
Source      Destination    IPV4Address    IPV6Address
-----      -----          -----          -----
EL-CID      8.8.8.8        8.8.8.8        8.8.8.8
                                         Bytes
                                         32

PS C:\Users> test-connection $myhost.name -count 1
Source      Destination    IPV4Address    IPV6Address
-----      -----          -----          -----
EL-CID      google-public... 8.8.8.8        8.8.8.8
                                         Bytes
                                         32

```

We may also add methods to the custom object as below. In this case we add a ping method to the host object. Within the scriptblock, the \$this variable is used to reference the current object.

```

Add-Member -inputobject $myhost -Membertype ScriptMethod -name ping ` 
    -value { param ( [parameter()] $count = 1 )
              test-connection $this.IPAddress -count $count
            }

```

```

PS C:\Users>
PS C:\Users> Add-Member -inputobject $myhost -Membertype ScriptMethod -name ping ` 
    -value { param ( [parameter()] $count = 1 )
              test-connection $this.IPAddress -count $count
            }
PS C:\Users>
PS C:\Users> $myhost.ping()
Source      Destination    IPV4Address    IPV6Address
-----      -----          -----          -----
EL-CID      8.8.8.8        8.8.8.8        8.8.8.8

PS C:\Users> $myhost.ping(2)
Source      Destination    IPV4Address    IPV6Address
-----      -----          -----          -----
EL-CID      8.8.8.8        8.8.8.8        8.8.8.8
EL-CID      8.8.8.8        8.8.8.8        8.8.8.8

PS C:\Users> $myhost.ip = '8.8.4.4'
PS C:\Users>
PS C:\Users> $myhost.ping()
Source      Destination    IPV4Address    IPV6Address
-----      -----          -----          -----
EL-CID      8.8.4.4        8.8.4.4        8.8.4.4

```

We may inspect the final object again using get-member.

```

PS C:\Users>
PS C:\Users> $myhost | get-member

TypeName: System.Management.Automation.PSCustomObject

Name      MemberType      Definition
----      -----      -----
ip        AliasProperty ip = IPAddress
name      AliasProperty name = HostName
Equals    Method         bool Equals(System.Object obj)
 GetHashCode Method       int GetHashCode()
 GetType   Method         type GetType()
 ToString  Method         string ToString()
 HostName NoteProperty   string HostName=google-public-dns-a.google.com
 IPAddress NoteProperty  string IPAddress=8.8.4.4
 ping     ScriptMethod   System.Object ping();

```

As mentioned earlier, one of the problems with this object is the inconsistency that is created when either the HostName or IPAddress property are changed. We need the HostName value to reflect the DNS name of the IP address stored in the property IPAddress and vice versa. To accomplish this we need a bit of cleverness.

We will create the object with two properties `_IP` and `_Name` each of which is of type `NoteProperty`. These two properties are used to hold values but are not set directly by the user. We then will create two `ScriptProperties` called `HostName` and `IPAddress`. As the name implies a `ScriptProperty` is a cross between a property and a method as we will see below. A `ScriptProperty` has a *Get* script block that retrieves the value of corresponding property either `_IP` or `_Name` and a *Set* script block the sets the value of the corresponding property either `_IP` or `_Name`.

```

$myhost = New-Object -TypeName PSObject

# These properties hold the values.
Add-Member -inputobject $myhost -NotePropertyName _IP ` 
 -NotePropertyValue ''

Add-Member -inputobject $myhost -NotePropertyName _Name ` 
 -NotePropertyValue ''

<#

```

The property HostName is a script the gets the value of `_Name` and sets the value of `_Name`. In addition when the HostName is changed, the set scriptblock also changes the IP address to reflect the resolved name

```
#>
Add-Member -inputobject $myhost -MemberType ScriptProperty -Name HostName `
    -value { # Get
        $this._Name
    } `
    { # Set
        param ( [string] $Name )
        $this._Name = $Name
        $ip = resolve-dnsname $Name -ErrorAction SilentlyContinue
        if ( $ip ) { $this._IP = $ip.IP4Address}
        else { $this._IP = '127.0.0.1' }
    }

<#
    The property IPAddress property works in a similar manner as the HostName
    property but for the IPAddress
#>
Add-Member -inputobject $myhost -MemberType ScriptProperty -Name IPAddress `
    -value { #Get
        $this._IP
    } `
    { #Set
        param ( [string] $IPAddr )
        $this._IP = $IPAddr
        $ip = resolve-dnsname $IPAddr -ErrorAction SilentlyContinue
        if ( $ip ) { $this._Name = $ip.NameHost}
        else { $this._Name = 'Unknown' }
    }
```

```

PS C:\Users>
PS C:\Users> $myhost.IPAddress

PS C:\Users> $myhost.HostName

PS C:\Users> $myhost.IPAddress = '8.8.8.8'
PS C:\Users>
PS C:\Users> $myhost.HostName
google-public-dns-a.google.com
PS C:\Users>
PS C:\Users> $myhost.HostName = 'www.msnbc.com'
PS C:\Users>
PS C:\Users> $myhost.IPAddress
23.221.3.10
PS C:\Users>

```

```

PS C:\Users>
PS C:\Users> $myhost | get-member

TypeName: System.Management.Automation.PSCustomObject

Name      MemberType      Definition
----      -----      -----
Equals    Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType   Method      type GetType()
ToString  Method      string ToString()
_IP      NoteProperty  string _IP=23.221.3.10
_Name    NoteProperty  string _Name=a23-221-3-10.deploy.static.akamaitechnologies.com
HostName  ScriptProperty System.Object HostName {get= # Get...
IPAddress ScriptProperty System.Object IPAddress {get= #Get...

```

The next script creates an collection of custom objects holding the IP address, host names, and Operating System for a set of hosts.

```

# these could come from a csv file
$ipaddr = @('23.235.44.73','8.8.4.4','8.8.8.8','23.77.237.50')

# create an empty array to hold the custom objects
[array] $hosts = $null

foreach ($ip in $ipaddr) {

    $ping = test-connection $ip -count 1
    if ($ping.ResponseTimeToLive -le 64 ) {
        $os = 'Linux'
    } elseif ($ping.ResponseTimeToLive -le 128) {
        $os = 'Windows'
    } else {

```

```
$os = 'Unix'
}

$dnsinfo = Resolve-DnsName $ip -ErrorAction SilentlyContinue
$dnsname = if ($dnsinfo) { $dnsinfo.server } else { 'Unknown' }

# Create a hash table properties of the object
$props = @{
    ipaddr=$ip
    hostname=$dnsname
    os=$os
}

# create the host object with the properties
$hostobj = New-Object -TypeName PSObject -Property $props
# append the object to the array
$hosts += $hostobj
}
```

```

2 # could come from a csv file
3 $ipaddr = @('23.235.44.73','8.8.4.4','8.8.8.8','23.77.237.50')
4
5 # create an empty array to hold the custom objects
6 [array] $hosts = $null
7
8 foreach ($ip in $ipaddr) {
9
10    $ping = test-connection $ip -count 1
11    if ($ping.ResponseTimeToLive -le 64 ) {
12        $os = 'Linux'
13    } elseif ($ping.ResponseTimeToLive -le 128) {
14        $os = 'Windows'
15    } else {
16        $os = 'Unix'
17    }
18
19    $dnsinfo = Resolve-DnsName $ip -ErrorAction SilentlyContinue
20    $dnsname = if ($dnsinfo) { $dnsinfo.server } else { 'Unknown' }
21
22    # Create a hash table properties of the object
23    $props = @{
24        ipaddr=$ip
25        hostname=$dnsname
26        os=$os
27    }
28
29    # create the host object with the properties
30    $hostobj = New-Object -TypeName PSObject -Property $props
31    # append the object to the array
32    $hosts += $hostobj
33 }

```

```
PS C:\Users> $hosts
```

ipaddr	hostname	os
-----	-----	--
23.235.44.73	Unknown	Linux
8.8.4.4	google-public-dns-b.google.com	Linux
8.8.8.8	google-public-dns-a.google.com	Linux
23.77.237.50	a23-77-237-50.deploy.static.akamaitechnologies.com	Linux

The host array created by the script may be used in a pipeline.

```
$hosts | foreach-object { test-connection $_.IPAddr -count 1}
```

```
PS C:\Users>
PS C:\Users> $hosts | foreach-object { test-connection $_.IPAddr -count 1}

Source          Destination        IPV4Address      IPV6Address
----          -----
EL-CID          23.235.44.73
EL-CID          8.8.4.4           8.8.4.4
EL-CID          8.8.8.8           8.8.8.8
EL-CID          23.77.237.50       23.77.237.50
```

The function below creates a new object from properties of multiple objects. This example is derived from one shown in "PowerShell in Depth", Jones, Siddaway, Hicks.

This example use CIM cmdlets instead of the traditional WMI. CIM has significant advantages over WMI including using the WS-MAN protocol instead of DCOM for establishing sessions with remote hosts.

```
function Get-MyHostObject ( $hostaddr) {

    if ($hostaddr -eq $null) {
        $h = 'localhost'
    } else {
        $h = $hostaddr
    }
    $os = Get-CimInstance -class Win32_OperatingSystem -ComputerName $h
    $cs = Get-CimInstance -class Win32_ComputerSystem -ComputerName $h
    $bios = Get-CimInstance -class Win32_BIOS -ComputerName $h
    $proc = Get-CimInstance -class Win32_Processor -ComputerName $h |
    Select -first 1

    $props = @{
        OSVersion=$os.version
        Model=$cs.model
        Manufacturer=$cs.manufacturer
        BIOSSerial=$bios.serialnumber
        ComputerName=$os.CSName
    }
}
```

```
OSArchitecture=$os.osarchitecture
ProcArchitecture=$proc.addresswidth}

return ( New-Object -TypeName PSObject -Property $props )
}
```

Extending an Object

Monday, February 4, 2013

5:09 PM

One of the powering features of modern programming languages is the ability extended a type by adding properties and methods. PowerShell supports extending types or adding new types through configuration changes. It also supports limited dynamic type extension. What this means is that an properties may be added to an existing PowerShell object.

The test-netconnection cmdlet is used for diagnostic testing of network connections.

```
# equivalent of a ping
test-netconnection 130.233.224.254

# trace route
test-netconnection 130.233.224.254 -traceroute

# test for an open port
test-netconnection 130.233.224.254 -port 80

test-netconnection 130.233.224.254 -CommonTCPPort http
```

```
PS C:\Users>
PS C:\Users> # equivalent of a ping
PS C:\Users> test-netconnection 130.233.224.254

ComputerName      : 130.233.224.254
RemoteAddress     : 130.233.224.254
InterfaceAlias    : vEthernet (LAN)
SourceAddress     : 192.168.15.202
PingSucceeded     : True
PingReplyDetails (RTT) : 161 ms


PS C:\Users>
PS C:\Users> # trace route
PS C:\Users> test-netconnection 130.233.224.254 -traceroute

ComputerName      : 130.233.224.254
RemoteAddress     : 130.233.224.254
InterfaceAlias    : vEthernet (LAN)
SourceAddress     : 192.168.15.202
PingSucceeded     : True
PingReplyDetails (RTT) : 150 ms
TraceRoute        :
  192.168.15.1
  10.0.0.2
  108.73.163.254
  76.205.15.1
  12.83.32.145
  12.122.133.33
  213.248.87.253
  62.115.113.30
  213.155.130.86
  213.248.85.174
  109.105.97.142
  109.105.97.138
  109.105.97.136
  109.105.97.131
  109.105.102.105
  TimedOut
  130.233.231.185
  130.233.224.254


PS C:\Users>
PS C:\Users> # test for an open port
PS C:\Users> test-netconnection 130.233.224.254 -port 80

ComputerName      : 130.233.224.254
RemoteAddress     : 130.233.224.254
RemotePort        : 80
InterfaceAlias    : vEthernet (LAN)
SourceAddress     : 192.168.15.202
PingSucceeded     : True
PingReplyDetails (RTT) : 155 ms
TcpTestSucceeded  : True


PS C:\Users>
PS C:\Users> test-netconnection 130.233.224.254 -CommonTCPPort http

ComputerName      : 130.233.224.254
RemoteAddress     : 130.233.224.254
RemotePort        : 80
```

The object returned by this cmdlet is of type TestNetConnectionResult. This object may be extended by adding properties that are missing and may be useful.

First, let's inspect the properties return by this cmdlet:

```
$t = test-netconnection 8.8.8.8  
$t | get-member
```

TypeName: TestNetConnectionResult		
Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
AllNameResolutionResults	Property	System.Object AllNameResolutionResults {
BasicNameResolution	Property	System.Object BasicNameResolution {get; set;}
ComputerName	Property	string ComputerName {get; set;}
Detailed	Property	bool Detailed {get; set;}
DNSOnlyRecords	Property	System.Object DNSOnlyRecords {get; set;}
InterfaceAlias	Property	string InterfaceAlias {get; set;}
InterfaceDescription	Property	string InterfaceDescription {get; set;}
InterfaceIndex	Property	uint32 InterfaceIndex {get; set;}
IsAdmin	Property	bool IsAdmin {get; set;}
LLMNRNetbiosRecords	Property	System.Object LLMNRNetbiosRecords {get; set;}
MatchingIPsecRules	Property	ciminstance[] MatchingIPsecRules {get; set;}
NameResolutionSucceeded	Property	bool NameResolutionSucceeded {get; set;}
NetAdapter	Property	ciminstance NetAdapter {get; set;}
NetRoute	Property	ciminstance NetRoute {get; set;}
NetworkIsolationContext	Property	string NetworkIsolationContext {get; set;}
PingReplyDetails	Property	System.Net.NetworkInformation.PingReply
PingSucceeded	Property	bool PingSucceeded {get; set;}
RemoteAddress	Property	ipaddress RemoteAddress {get; set;}
RemotePort	Property	uint32 RemotePort {get; set;}
SourceAddress	Property	ciminstance SourceAddress {get; set;}
TcpClientSocket	Property	System.Net.Sockets.Socket TcpClientSocket
TcpTestSucceeded	Property	bool TcpTestSucceeded {get; set;}
TraceRoute	Property	string[] TraceRoute {get; set;}

Let's extend the object returned by test-netconnection by add a HostName property that shows the name of the remote host provided the host is registered in DNS.

```

function MyTest-NetConnection ($hostaddr) {
    # get the information
    $hostobj = test-netconnection $hostaddr
    # get the name of the remote host
    $dnsinfo = Resolve-DnsName $hostaddr

    # add the HostName property
    Add-Member -inputobject $hostobj -NotePropertyName 'HostName' -
    NotePropertyValue $dnsinfo.server

    return $hostobj
}

$result = MyTest-NetConnection 8.8.8.8

```

```

PS C:\Users>
PS C:\Users> function MyTest-NetConnection ($hostaddr) {
>>>     # get the information
>>>     $hostobj = test-netconnection $hostaddr
>>>     # get the name of the remote host
>>>     $dnsinfo = Resolve-DnsName $hostaddr
>>>
>>>     # add the HostName property
>>>     Add-Member -inputobject $hostobj -NotePropertyName 'HostName' -NotePropertyValue
>>>
>>>     return $hostobj
>>> }
PS C:\Users>
PS C:\Users> $result = MyTest-NetConnection 8.8.8.8
PS C:\Users>
PS C:\Users> $result

ComputerName      : 8.8.8.8
RemoteAddress     : 8.8.8.8
InterfaceAlias    : vEthernet (LAN)
SourceAddress     : 192.168.15.202
PingSucceeded     : True
PingReplyDetails (RTT) : 29 ms

PS C:\Users> $result | format-table RemoteAddress, HostName

```

RemoteAddress	HostName
8.8.8.8	google-public-dns-a.google.com

Inspecting the object we see that the object is of type TestNetConnectionResult with the additional HostName property.

TypeName: TestNetConnectionResult		
Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
HostName	NoteProperty	string HostName=google-public-dns-a.google.
AllNameResolutionResults	Property	System.Object AllNameResolutionResults {get;
BasicNameResolution	Property	System.Object BasicNameResolution {get;set;
ComputerName	Property	string ComputerName {get;set;}
Detailed	Property	bool Detailed {get;set;}
DNSOnlyRecords	Property	System.Object DNSOnlyRecords {get;set;}
InterfaceAlias	Property	string InterfaceAlias {get;set;}
InterfaceDescription	Property	string InterfaceDescription {get;set;}
InterfaceIndex	Property	uint32 InterfaceIndex {get;set;}
IsAdmin	Property	bool IsAdmin {get;set;}
LLMNRNetbiosRecords	Property	System.Object LLMNRNetbiosRecords {get;set;}
MatchingIPsecRules	Property	ciminstance[] MatchingIPsecRules {get;set;}
NameResolutionSucceeded	Property	bool NameResolutionSucceeded {get;set;}
NetAdapter	Property	ciminstance NetAdapter {get;set;}
NetRoute	Property	ciminstance NetRoute {get;set;}
NetworkIsolationContext	Property	string NetworkIsolationContext {get;set;}
PingReplyDetails	Property	System.Net.NetworkInformation.PingReply Ping
PingSucceeded	Property	bool PingSucceeded {get;set;}
RemoteAddress	Property	ipaddress RemoteAddress {get;set;}
RemotePort	Property	uint32 RemotePort {get;set;}
SourceAddress	Property	ciminstance SourceAddress {get;set;}
TcpClientSocket	Property	System.Net.Sockets.Socket TcpClientSocket {
TcpTestSucceeded	Property	bool TcpTestSucceeded {get;set;}
TraceRoute	Property	string[] TraceRoute {get;set;}

Connecting

Tuesday, April 23, 2013
7:28 AM

Active Directory is a hierarchical database of information about network users and resources. AD implements Lightweight Directory Access Protocol (LDAP) for managing the AD database.

For all the methods illustrated for querying or altering AD object properties, it's important to understand that AD does not return an object attribute if the value

of that attribute is null. This may lead to errors being thrown when the get method is used to retrieve an attribute value which is null.

There are four methods for connecting to Active Directory:

1. The .Net DirectoryServices class
2. The [ADSI] accelerator
3. The Active Directory PSProvider
4. Active Directory cmdlets

The .Net DirectoryServices class is used when the host on the script is running is not a member of the domain. The example below shows a connection using the administrator credentials to the server identified by 10.10.10.100 running the Active Directory Domain Services where cis161-100.local is the root of the AD tree,

```
$ADRoot = New-Object DirectoryServices.DirectoryEntry(  
    "LDAP://10.10.10.100",  
    "CIS161-100.local\administrator",  
    "cntpword")
```

```
PS C:\Users>  
PS C:\Users> $ADRoot = New-Object DirectoryServices.DirectoryEntry(  
>     "LDAP://10.10.10.100",  
>     "CIS161-100.local\administrator",  
>     "cntpword")  
>  
PS C:\Users>  
PS C:\Users> $adroot  
  
distinguishedName : {DC=CIS161-100,DC=local}  
Path             : LDAP://10.10.10.100
```

If the host on which the PS script is running is a member of the domain the [ADSI] accelerator is the preferred method for connecting to the AD. The ADSI accelerator always authenticates using the current identity. There identify of the

AD Domain Services server is specifically inferred from the AD connection. The example below works on a host that is part of a domain or on a server in the domain.

```
$ADRoot = [ADS1] ""
```

```
PS C:\Users>
PS C:\Users> $ADRoot = [ADS1] ""
PS C:\Users>
PS C:\Users> $adroot

distinguishedName : {DC=cis161-100,DC=local}
Path               :
```

The third method for connecting is the Active Directory PSProvider. This is more limited than the other two when for acting upon the directory but much easier for navigating the directory. This requires that the *Active Directory* module be loaded in the current PowerShell session. The ActiveDIrectory module for PowerShell is available in "Remote Server Administration Tools". This package also provides PowerShell cmdlets for managing Active Directory. This module is installed by default on Windows Server but not on Windows workstation. If remote administration using the Active Directory cmdlets is required, the RSAT package must be installed.

The syntax below creates a new PSDrive called myAD. After that command executes, the user may navigated the AD tree using conventional file system navigation commands (set-location alias cd) and inspection (get-childitem alias dir) techniques.

```
import-module activedirectory

new-psdrive -name myAD -PSProvider ActiveDirectory -Root "dc=cis161-100,dc=local"
```

In the above, the new-psdrive cmdlet authenticates using the current identity, however, optional credentials may be provided.

Navigating the AD Tree

Tuesday, November 12, 2013
3:04 PM

Similar to other stores, the Active Directory tree may be abstracted using the ActiveDirectory PS Provider. This allows the use of conventional navigational and inspection cmdlets.

The ActiveDirectory PSProvider is not normally available but must be imported into the current session as shown below. This module is normally available on Server 2012 and above. Workstations require the "Remote Server Administration Tools" update available from Microsoft.

```
import-module ActiveDirectory
```

This makes both the provider and AD cmdlets available. Load this module on the server makes the PSdrive AD: available. The AD: drive allows the user to navigate the Active Directory tree using the usual cmdlets.

```
PS C:\Users>
PS C:\Users> import-module ActiveDirectory
PS C:\Users>
PS C:\Users> set-location ad:
PS AD:>
PS AD:> get-childitem
```

Name	ObjectClass	DistinguishedName
CIS161-100	domainDNS	DC=CIS161-100,DC=local
Configuration	configuration	CN=Configuration,DC=CIS161-100,DC=local
Schema	dMD	CN=Schema,CN=Configuration,DC=CIS161-100,DC=local
DomainDnsZones	domainDNS	DC=DomainDnsZones,DC=CIS161-100,DC=local
ForestDnsZones	domainDNS	DC=ForestDnsZones,DC=CIS161-100,DC=local

On a workstation we may also create a new PSDrive using the syntax show below. We may then navigate the remote ActiveDIrectory tree using the usual cmdlets. If

the workstation is not part of the AD domain then credentials may need to be provided using the -credential parameter.

```
new-psdrive -name myAD -PSProvider ActiveDirectory -Root "dc=cis161-100,dc=local"  
-server 10.10.10.100
```

```
PS C:\Users>  
PS C:\Users> new-psdrive -name myAD -PSProvider ActiveDirectory '  
>> -Root "dc=cis161-100,dc=local" -server 10.10.10.100  


| Name | Used (GB) | Free (GB) | Provider      | Root                             |
|------|-----------|-----------|---------------|----------------------------------|
| myAD |           |           | ActiveDire... | //RootDSE/dc=cis161-100,dc=local |

  
PS C:\Users> set-location myAD:  
PS myAD:>  
PS myAD:> get-childitem  


| Name                 | ObjectClass          | DistinguishedName                                   |
|----------------------|----------------------|-----------------------------------------------------|
| Builtin              | builtinDomain        | CN=Builtin,DC=CIS161-100,DC=local                   |
| Computers            | container            | CN=Computers,DC=CIS161-100,DC=local                 |
| Domain Controllers   | organizationalUnit   | OU=Domain Controllers,DC=CIS161-100,DC=local        |
| ForeignSecurityPr... | container            | CN=ForeignSecurityPrincipals,DC=CIS161-100,DC=local |
| Infrastructure       | infrastructureUpdate | CN=Infrastructure,DC=CIS161-100,DC=local            |
| LostAndFound         | lostAndFound         | CN=LostAndFound,DC=CIS161-100,DC=local              |
| Managed Service A... | container            | CN=Managed Service Accounts,DC=CIS161-100,DC=local  |
| NTDS Quotas          | msDS-QuotaContainer  | CN=NTDS Quotas,DC=CIS161-100,DC=local               |
| Program Data         | container            | CN=Program Data,DC=CIS161-100,DC=local              |
| System               | container            | CN=System,DC=CIS161-100,DC=local                    |
| TPM Devices          | msTPM-Information... | CN=TPM Devices,DC=CIS161-100,DC=local               |
| Users                | container            | CN=Users,DC=CIS161-100,DC=local                     |


```

Navigating to the various Active Directory containers requires slightly different syntax than normal. The example below shows navigation to the users container using a relative distinguished name that identifies the *Common Name*.

```
set-location "cn=users"
```

```

PS myAD:\>
PS myAD:\> set-location "cn=users"
PS myAD:\cn=users>
PS myAD:\cn=users> get-childitem

Name          ObjectClass      DistinguishedName
----          -----
Administrator   user           CN=Administrator,CN=Users,DC=CIS161-100,DC=1
Allowed RODC Pass... group        CN=Allowed RODC Password Replication Group,C
Cert Publishers group        CN=Cert Publishers,CN=Users,DC=CIS161-100,DC=
Cloneable Domain ... group       CN=Cloneable Domain Controllers,CN=Users,DC=
Denied RODC Passw... group        CN=Denied RODC Password Replication Group,CN=
DnsAdmins      group        CN=DnsAdmins,CN=Users,DC=CIS161-100,DC=local
DnsUpdateProxy group        CN=DnsUpdateProxy,CN=Users,DC=CIS161-100,DC=
Domain Admins   group        CN=Domain Admins,CN=Users,DC=CIS161-100,DC=1
Domain Computers group        CN=Domain Computers,CN=Users,DC=CIS161-100,D
Domain Controllers group        CN=Domain Controllers,CN=Users,DC=CIS161-100
Domain Guests    group        CN=Domain Guests,CN=Users,DC=CIS161-100,DC=1
Domain Users     group        CN=Domain Users,CN=Users,DC=CIS161-100,DC=lo
Enterprise Admins group        CN=Enterprise Admins,CN=Users,DC=CIS161-100,
Enterprise Read-o... group       CN=Enterprise Read-only Domain Controllers,C
Group Policy Crea... group        CN=Group Policy Creator Owners,CN=Users,DC=C
Guest           user           CN=Guest,CN=Users,DC=CIS161-100,DC=local
krbtgt          user           CN=krbtgt,CN=Users,DC=CIS161-100,DC=local

```

Similarly navigating to the "Domain Controllers" container , which is an *Organizational Unit*, requires a distinguished name identifying the container as an OU. In the example below, a relative distinguished name is used that navigates up to the parent container then down to the OU.

```
set-location ..\\"ou=Domain Controllers"
```

```

PS myAD:\cn=users>
PS myAD:\cn=users> set-location ..\\"ou=Domain Controllers"
PS myAD:\ou=Domain Controllers>
PS myAD:\ou=Domain Controllers> gci

Name          ObjectClass      DistinguishedName
----          -----
CIS161-100    computer        CN=CIS161-100,OU=Domain Controller

```

Enumerating objects in the AD tree is similar what is used for other PSDrives. The get-childitem cmdlet is used to enumerate the objects. In this example, we navigate to the root of the AD tree the use get-childitem with the -recurse switch to retrieve all AD objects. Those objects are then piped in where-object where only user objects are selected.

```
get-childitem -recurse | where-object { $_.objectclass -eq 'user' }
```

```
PS myAD:\ou=Domain Controllers>
PS myAD:\ou=Domain Controllers> set-location \
PS myAD:\>
PS myAD:\> get-childitem -recurse |where-object { $_.objectclass -eq 'user' }

Name          ObjectClass      DistinguishedName
----          -----
Administrator user           CN=Administrator,CN=Users,DC=CIS161-100,DC=Local
Guest         user           CN=Guest,CN=Users,DC=CIS161-100,DC=Local
mgalea        user           CN=mgalea,CN=Users,DC=CIS161-100,DC=Local
krbtgt        user           CN=krbtgt,CN=Users,DC=CIS161-100,DC=Local
```

AD Users and Groups

Tuesday, April 23, 2013
2:30 PM

To simplify using PowerShell with Active Directory, Microsoft developed an Active Directory module that may be imported into the PowerShell session. This module contains many cmdlets for working with AD.

Server 2012 has a tile on the Start Screen to PowerShell that automatically loads this module.

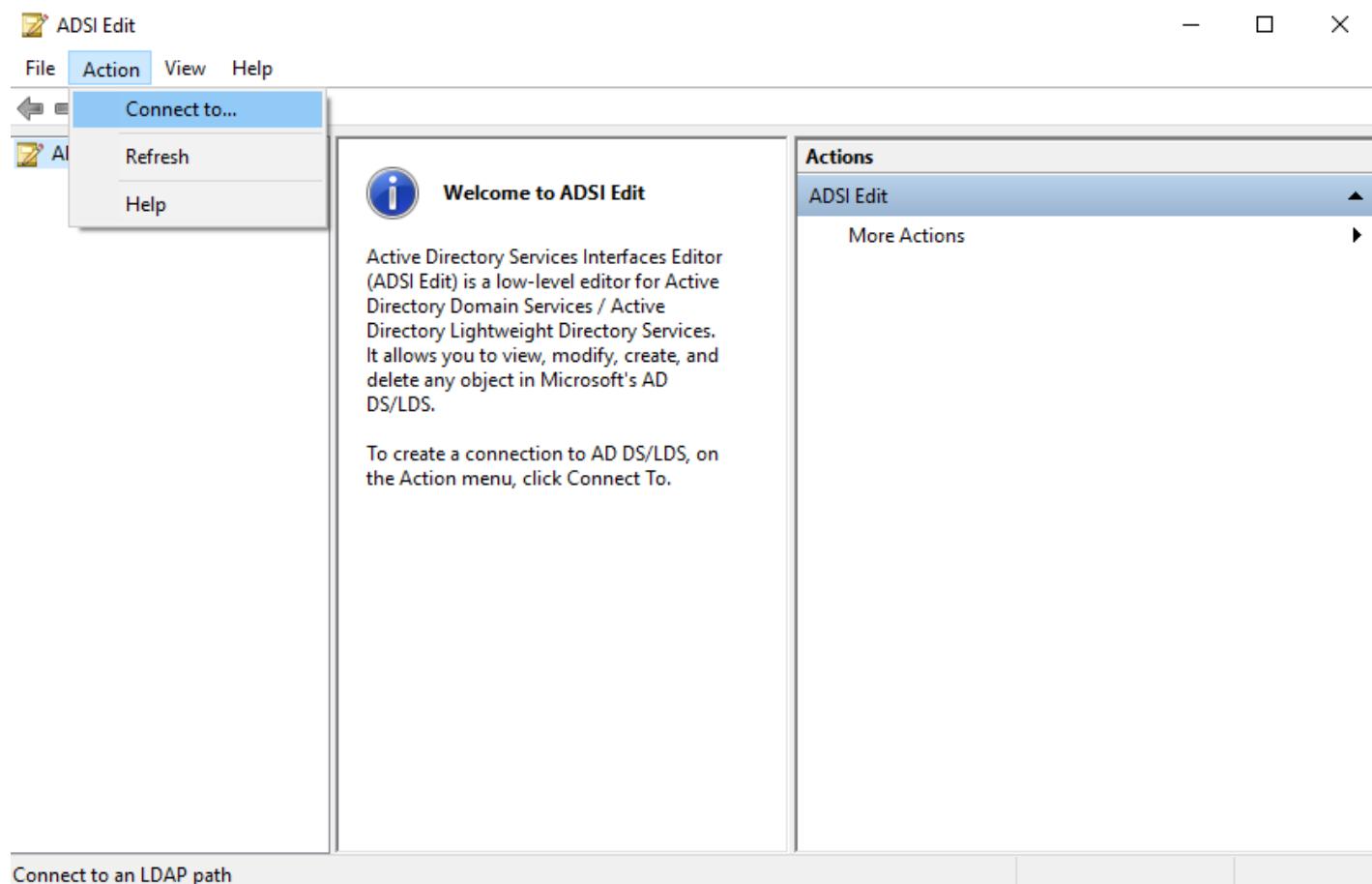
The cmdlets shown below require the installation of the Remote Server Administration Tools (RSAT) and the Active Directory module loaded into the current session.

The cmdlets available in this module are documented at:
<http://technet.microsoft.com/en-us/library/ee617195.aspx>

Querying Users:

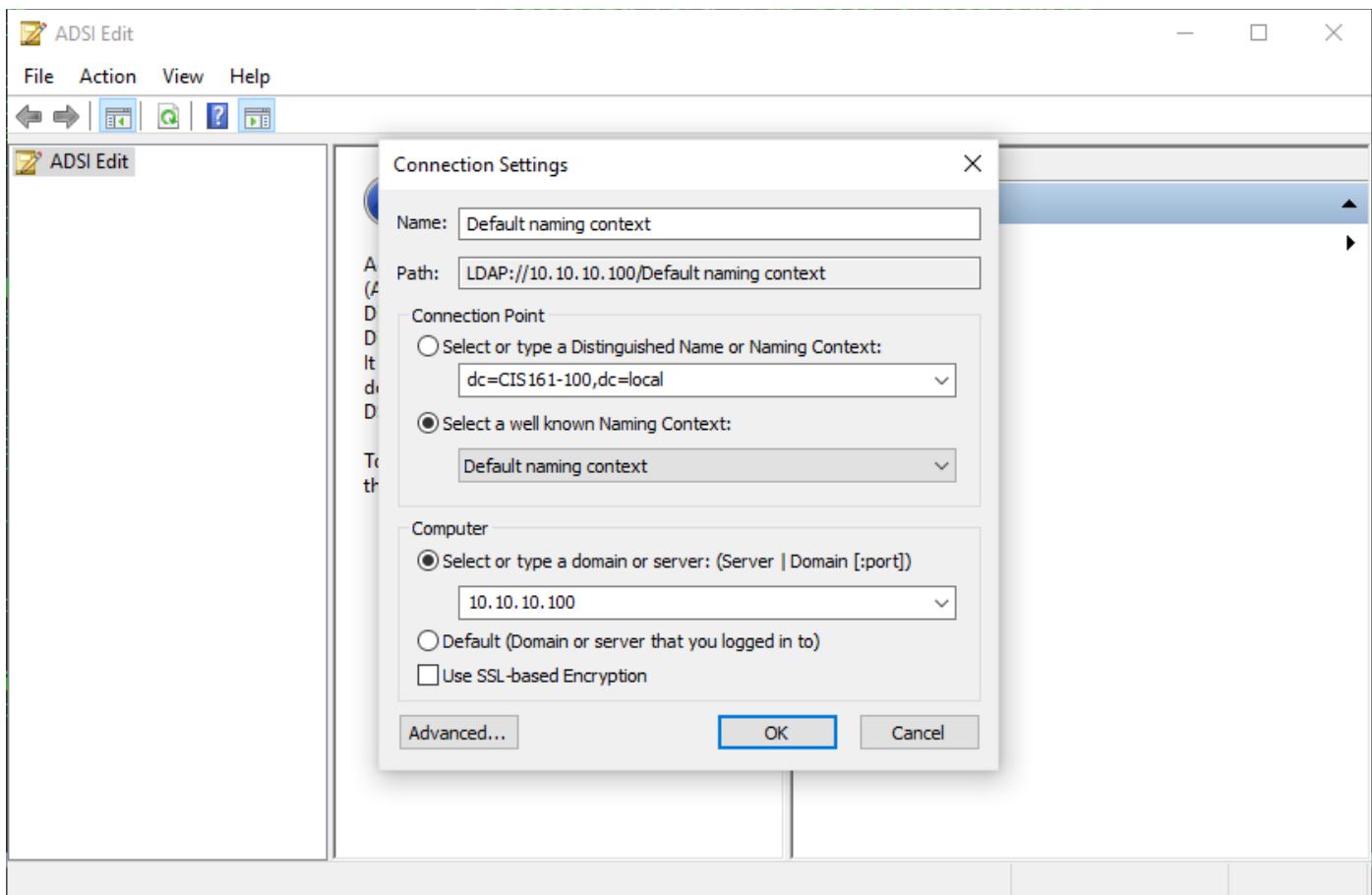
Some of the examples below specify specific user object properties. The property names are not the same as the labels used in the "Active Directory Users and Computers" GUI console. To discover the property names, we may query user objects as shown in the examples below listing all properties. We may also use the ADSIEdit console which is natively available Server 2012 or installed on the workstation when the RSAT package is installed.

In example, ADSIEdit is executed on a workstation with RSAT installed. ADSIEdit may be started from the command line. Select Action -> Connect to open the connection dialog box.



The LDAP information must entered in the format shown below. When running ADSIEdit remotely click Advanced button and enter the administer credentials

before attempting to connect. This step is not necessary when executing ADSIEdit on the Domain Controller.



We may now drill down to a user object to expose the properties.

ADSI Edit

File Action View Help

Default naming context [10.1] DC=CIS161-100,DC=local

Name	Class
OU=AnnArbor	organization...
CN=Builtin	builtinDomain
CN=Computers	container
OU=Domain Controllers	organization...
CN=ForeignSecurityPrincip...	container
CN=LostAndFound	lostAndFound
CN=Managed Service Acco...	container
CN=NTDS Quotas	msDS-Quota...
CN=Program Data	container
CN=System	container
CN=TPM Devices	msTPM-Info...
CN=Users	container
CN=Infrastructure	infrastructur...

Actions

DC=CIS161-100,DC=local

More Actions

In the rightmost pane under the user object, select "More Actions" then select "Properties"

ADSI Edit

File Action View Help

Default naming context [10.1]

DC=CIS161-100,DC=local

OU=AnnArbor

CN=Builtin

CN=Computers

OU=Domain Control

CN=ForeignSecurity

CN=LostAndFound

CN=Managed Services

CN=NTDS Quotas

CN=Program Data

CN=System

CN=TPM Devices

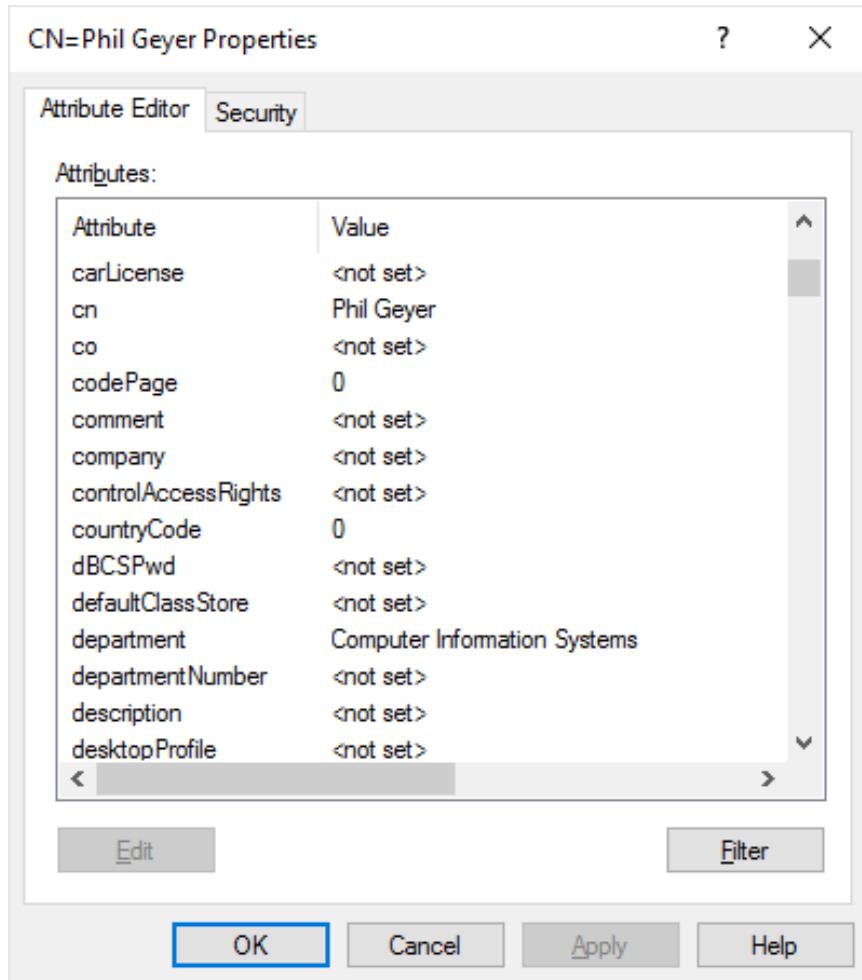
CN=Users

Name	Class	Actions
CN=Phil Geyer	user	OU=AnnArbor More Actions
CN=Phil Geyer	user	More Actions

The user object properties are shown in a separate window.

The screenshot shows the ADSI Edit interface. On the left is a tree view of the directory structure under 'Default naming context'. A user object 'CN=Phil Geyer' is selected. The main pane displays a table with columns 'Name', 'Class', and 'Actions'. The 'Name' column shows 'CN=Phil Geyer' twice, and the 'Class' column shows 'user' both times. The 'Actions' column contains 'OU=AnnArbor' and 'More Actions' for the first entry, and 'More Actions' for the second entry. At the bottom, there are navigation buttons for the tree view and the main pane.

The user object properties are shown in a separate window.



The queries below use the properties identified using the above property window.

Depending upon what the user is attempting to accomplish, the cmdlet requires a unique account identity. The account identity is identified by the `-identity` parameter. The value for this parameter may be the `SAMAccountName`, i.e., the username used at logon, or distinguished name. The examples below use both. Generally speaking depending upon the action being taken to eliminate ambiguity, the identity needs to be specific which in this case implies using a distinguished name.

All users in the AD tree may be queried using the syntax below. This query retrieves the default set of properties of the user object.

```
get-aduser -filter *
```

```
PS myAD:>
PS myAD:> get-aduser -filter *

DistinguishedName : CN=Administrator,CN=Users,DC=CIS161-100,DC=local
Enabled          : True
GivenName        :
Name              : Administrator
ObjectClass      : user
ObjectGUID       : a9cc0572-5a19-4394-9e8e-3da5f59b713f
SamAccountName   : Administrator
SID              : S-1-5-21-2841689656-2999035272-1982814355-500
Surname          :
UserPrincipalName :

DistinguishedName : CN=Guest,CN=Users,DC=CIS161-100,DC=local
Enabled          : False
GivenName        :
Name              : Guest
ObjectClass      : user
ObjectGUID       : d71ea37f-8cef-486c-9874-5c9df0c3774d
SamAccountName   : Guest
SID              : S-1-5-21-2841689656-2999035272-1982814355-501
Surname          :
UserPrincipalName :

DistinguishedName : CN=mailuser,CN=Users,DC=CIS161-100,DC=local
```

To retrieve all properties of the user object the -properties switch parameter may be used as in the examples below.

```
get-aduser -filter * -properties * | out-host -paging
```

```
PS myAD:\>
PS myAD:\> get-aduser -filter * -properties * | out-host -paging

AccountExpirationDate          : 0
accountExpires                 :
AccountLockoutTime             : False
AccountNotDelegated            : 1
adminCount                      : False
AllowReversiblePasswordEncryption : {}
AuthenticationPolicy            : {}
AuthenticationPolicySilo        : {}
BadLogonCount                  : 0
badPasswordTime                : 130916508953634136
badPwdCount                     : 0
CannotChangePassword           : False
CanonicalName                  : CIS161-100.local/Users/Administrato
Certificates                   : {}
City                           :
CN                            : Administrator
codePage                       : 0
Company                        :
CompoundIdentitySupported      : {}
Country                         :
countryCode                     : 0
Created                         : 3/30/2015 4:19:50 PM
createTimeStamp                 : 3/30/2015 4:19:50 PM
Deleted                         :
Department                      :
Description                     : Built-in account for administering
displayName                     :
```

To select a specific user(s) requires using the `-filter` parameter with a conditional expression. The `-filter` parameter requires PowerShell conditional expressions. In this example, we select all users with the surname of geyer.

```
get-aduser -filter { (sn -eq 'geyer') } -properties *
```

```
PS myAD:\>
PS myAD:\> get-aduser -filter {($sn -eq 'geyer')} -properties *

AccountExpirationDate      :
accountExpires              :
AccountLockoutTime          :
AuthenticationPolicy         : {}
AuthenticationPolicySilo     : {}
BadLogonCount                :
CannotChangePassword        : False
CanonicalName                :
Certificates                 : {}
City                         : Phil Geyer
CN                           : 0
codePage                     :
Company                      :
CompoundIdentitySupported    : {}
Country                      :
countryCode                  : 0
Created                      :
Deleted                      :
Department                   : Computer Information Systems
Description                  :
DisplayName                  : Phil Geyer
DistinguishedName           : CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=local
Division                     :
EmailAddress                : pgeyer@gmail.com
EmployeeID                  :
EmployeeType                :
```

Since the user object has many properties, we may limit the properties returned using the -properties parameter.

```
get-aduser -identity pgeyer -properties DisplayName, Department,
DistinguishedName
```

```
PS myAD:\>
PS myAD:\>
PS myAD:\> get-aduser -identity pgeyer -properties DisplayName, Department, DistinguishedName

Department      : Computer Information Systems
DisplayName     : Phil Geyer
DistinguishedName: CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=local
GivenName       : Phil
Name            : Phil Geyer
ObjectClass     : user
ObjectGUID      : 92892c2a-f87d-47ab-9809-67d08b49a8dc
SamAccountName  : pgeyer
SID             : S-1-5-21-2841689656-2999035272-1982814355-1105
Surname         : Geyer
UserPrincipalName: pgeyer@CIS161-100.local
```

A full distinguished name may be used with the identify parameter.

```
get-aduser -identity "cn=Phil Geyer,ou=AnnArbor,dc=cis161-100,dc=local"
```

```
PS myAD:\>
PS myAD:\> get-aduser -identity "cn=Phil Geyer,ou=AnnArbor,dc=cis161-100,dc=local"

DistinguishedName : CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=local
GivenName          : Phil
Name               : Phil Geyer
ObjectClass        : user
ObjectGUID         : 92892c2a-f87d-47ab-9809-67d08b49a8dc
SamAccountName    : pgeyer
SID                : S-1-5-21-2841689656-2999035272-1982814355-1105
Surname            : Geyer
UserPrincipalName : pgeyer@CIS161-100.local
```

We may also use the SAMaccountname as below to query the user properties/

```
get-aduser -identity pgeyer
```

This example uses an LDAP filter to query a specific user object. More

information about LDAP filter syntax is found at

<http://social.technet.microsoft.com/wiki/contents/articles/5392.active-directory-ldap-syntax-filters.aspx>.

```
get-aduser -LDAPfilter "(name=Phil Geyer)" -searchscope subtree -
searchbase "ou=AnnArbor,dc=cis161-100,dc=local"
```

```
PS myAD:\>
PS myAD:\> get-aduser -LDAPfilter "(name=Phil Geyer)" -searchscope subtree -searchbase "dc=cis161-100,dc=local"

DistinguishedName : CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=Local
GivenName          : Phil
Name               : Phil Geyer
ObjectClass        : user
ObjectGUID         : 92892c2a-f87d-47ab-9809-67d08b49a8dc
SamAccountName    : pgeyer
SID                : S-1-5-21-2841689656-2999035272-1982814355-1105
Surname            : Geyer
UserPrincipalName : pgeyer@CIS161-100.local
```

This example queries all users starting at the Active Directory root.

```
get-aduser -LDAPfilter "(ObjectClass=user)" -searchscope subtree -searchbase "dc=cis161-100,dc=local"
```

```
PS myAD:\>
PS myAD:\> get-aduser -LDAPfilter "(ObjectClass=user)" -searchscope subtree -searchbase "dc=cis161-100,dc=local"

DistinguishedName : CN=Administrator,CN=Users,DC=CIS161-100,DC=Local
Enabled           : True
GivenName          :
Name               : Administrator
ObjectClass        : user
ObjectGUID         : a9cc0572-5a19-4394-9e8e-3da5f59b713f
SamAccountName    : Administrator
SID                : S-1-5-21-2841689656-2999035272-1982814355-500
Surname            :
UserPrincipalName :

DistinguishedName : CN=Guest,CN=Users,DC=CIS161-100,DC=Local
Enabled           : False
GivenName          :
Name               : Guest
ObjectClass        : user
ObjectGUID         : d71ea37f-8cef-486c-9874-5c9df0c3774d
SamAccountName    : Guest
SID                : S-1-5-21-2841689656-2999035272-1982814355-501
Surname            :
UserPrincipalName :
```

This example retrieves the groups to which the user belongs.

```
$u = get-aduser -LDAPfilter "(name=Phil Geyer)" -searchscope subtree -searchbase "ou=AnnArbor,dc=cis161-100,dc=local" -properties *
```

```
$u.memberof
```

```
PS myAD:\>
PS myAD:\> $u = get-aduser -LDAPfilter "(name=Phil Geyer)" -searchscope subtree
PS myAD:\> $u.memberof
CN=Linux_Fans,OU=AnnArbor,DC=CIS161-100,DC=local
```

We may use a pipeline when searching for users. This example retrieves all members of the Computer Information Systems Department.

```
get-aduser -filter * -properties * |
    where-object { $_.Department -like '*Comp*Info*' }
```

```
PS myAD:\>
PS myAD:\> get-aduser -filter * -properties * |
>>     where-object { $_.Department -like '*Comp*Info*' }

AccountExpirationDate : 9223372036854775807
accountExpires          :
AccountLockoutTime      :
AccountNotDelegated     : False
AllowReversiblePasswordEncryption : False
AuthenticationPolicy      : {}
AuthenticationPolicySilo   : {}
BadLogonCount            : 0
badPasswordTime          : 0
badPwdCount              : 0
CannotChangePassword     : False
CanonicalName            : CIS161-100.local/AnnArbor/Phil Geyer
Certificates             : {}
City                     : Ann Arbor
CN                      : Phil Geyer
codePage                 : 0
Company                  :
CompoundIdentitySupported : {}
Country                 :
```

Updating a user property:

[https://technet.microsoft.com/en-us/library/hh852287\(v=wps.630\).aspx](https://technet.microsoft.com/en-us/library/hh852287(v=wps.630).aspx)

The set-aduser cmdlet is used to update properties other than passwords and other security related properties.

```

set-aduser -identity 'cn=Phil Geyer,ou=AnnArbor,dc=cis161-100,dc=local' ` 
    -office BE230 ` 
    -StreetAddress "4800 East Huron River Drive" ` 
    -state MI

get-aduser -LDAPfilter "(name=Phil Geyer)" -searchscope subtree - 
    searchbase "ou=AnnArbor,dc=cis161-100,dc=local" ` 
    -properties name, office, streetaddress, state

```

```

PS myAD:\>
PS myAD:\> set-aduser -identity 'cn=Phil Geyer,ou=AnnArbor,dc=cis161-100,dc=local' ` 
>>     -office BE230 ` 
>>     -StreetAddress "4800 East Huron River Drive" ` 
>>     -state MI
PS myAD:\>
PS myAD:\> get-aduser -LDAPfilter "(name=Phil Geyer)" -searchscope subtree -searchbase "all" ` 
>>     -properties name, office, streetaddress, state

DistinguishedName : CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=local
Enabled           : True
GivenName         : Phil
Name              : Phil Geyer
ObjectClass       : user
ObjectGUID        : 92892c2a-f87d-47ab-9809-67d08b49a8dc
Office            : BE230
SamAccountName   : pgeyer
SID               : S-1-5-21-2841689656-2999035272-1982814355-1105
State             : MI
StreetAddress     : 4800 East Huron River Drive
Surname          : Geyer
UserPrincipalName : pgeyer@CIS161-100.local

```

One of the issues with the set-ADUser cmdlet is that a result is not returned indicating the disposition of the update attempt. One of the ways to handle errors is *try-catch*. The example below illustrates one possible solution. In this case an invalid identity is specified.

```

$identity = 'jdoe'
try {
    set-aduser -identity $identity ` 
        -office BE230 ` 
        -StreetAddress "4800 East Huron River Drive" ` 
        -state MI ` 
        -erroraction SilentlyContinue
} catch {
    write-host "User $identity not found" -foregroundcolor red
}

```

```

PS myAD:>
PS myAD:> $identity = 'jdoe'
PS myAD:> try {
>>     set-aduser -identity $identity
>>         -office BE230
>>         -StreetAddress "4800 East Huron River Drive"
>>         -state MI
>>         -erroraction SilentlyContinue
>> } catch {
>>     write-host "User $identity not found" -foregroundcolor red
>>
>> }
User jdoe not found

```

The `-replace` parameter may be used with a hash table to update multiple properties. However, this requires knowing the corresponding LDAP attribute name for the property. This execution is updates the same properties and the previous one. Note the use of attribute names in the hash table.

```

$identity = 'pgeyer'
try {
    set-aduser -identity $identity ` 
        -replace @{PhysicalDeliveryOfficeName="B230";
                   StreetAddress="4800 East Huron River Drive";
                   st="MI"} -erroraction SilentlyContinue
    write-host "Attributes for User $identity updated" -foregroundcolor yellow
} catch {
    write-host "User $identity not found" -foregroundcolor red
}

```

```

PS myAD:>
PS myAD:> $identity = 'pgeyer'
PS myAD:> try {
>>     set-aduser -identity $identity ` 
>>         -replace @{PhysicalDeliveryOfficeName="B230";
>>                     StreetAddress="4800 East Huron River Drive";
>>                     st="MI"} -erroraction SilentlyContinue
>>     write-host "Attributes for User $identity updated" -foregroundcolor yellow
>> } catch {
>>     write-host "User $identity not found" -foregroundcolor red
>> }
Attributes for User pgeyer updated

```

We may also use a pipeline to update account information. Many of the above examples may be written to use a pipeline. A few examples on using a pipeline follow.

```
$identity = 'pgeyer'  
get-aduser -identity $identity | set-aduser -city 'Ann Arbor'
```

```
PS myAD:\>  
PS myAD:\> $identity = 'pgeyer'  
PS myAD:\> get-aduser -identity $identity | set-aduser -city 'Ann Arbor'  
PS myAD:\>
```

When multiple changes need to be made to a user or multiple users, it is more efficient to first *get* object in a variable, update the variable, then use *set-aduser* with the *-instance* parameter. This is illustrated in the script below.

```
$identity = 'pgeyer'  
$user = get-aduser -identity $identity  
$user.StreetAddress = ' 4800 East Huron River Drive'  
$user.State = 'MI'  
$user.PostalCode = '48105'  
$user.Office = 'BE230'  
$user.Department = 'Computer Information Systems'  
set-aduser -instance $user
```

```
PS myAD:\>  
PS myAD:\> $identity = 'pgeyer'  
PS myAD:\> $user = get-aduser -identity $identity  
PS myAD:\> $user.StreetAddress = ' 4800 East Huron River Drive'  
PS myAD:\> $user.State = 'MI'  
PS myAD:\> $user.PostalCode = '48105'  
PS myAD:\> $user.Office = 'BE230'  
PS myAD:\> $user.Department = 'Computer Information Systems'  
PS myAD:\> set-aduser -instance $user  
PS myAD:\>
```

Working with account passwords:

The script below prompts for a username and a new password and updates the Active Directory password.

```
$user = read-host 'Enter user name (SAMaccountname)'
```

```

$passwd = read-host "Enter new password for $user" -AsSecurestring
try {
    Set-ADAccountPassword -identity $user -NewPassword $passwd
    write-host "Password updated successfully for user $user" -
foregroundcolor yellow
} catch {
    write-host "Problem in updating password for user $user" -
foregroundcolor red
}

```

```

PS myAD:\>
PS myAD:\> $user = read-host 'Enter user name (SAMaccountname)'
Enter user name (SAMaccountname): pgeyer
PS myAD:\> $passwd = read-host "Enter new password for $user" -AsSecurestring
Enter new password for pgeyer: *****
PS myAD:\> try {
>>     Set-ADAccountPassword -identity $user -NewPassword $passwd
>>     write-host "Password updated successfully for user $user" -foregroundcolor yellow
>> } catch {
>>     write-host "Problem in updating password for user $user" -foregroundcolor red
>> }
Password updated successfully for user pgeyer

```

The script below forces a password change at the next login of the designated user. The *PasswordNeverExpires* property must be set to \$false to ensure that the *ChangePasswordAtLogon* property can be set to \$true.

```

$user = read-host 'Enter user name (SAMaccountname)'
try {
    Set-ADUser -identity $user -PasswordNeverExpires $False
    Set-ADUser -identity $user -ChangePasswordAtLogon $True
    write-host "Attribute updated successfully for user $user" -
foregroundcolor yellow
} catch {
    write-host "Problem in updating attribute for user $user" -
foregroundcolor red
}

```

```

PS myAD:\>
PS myAD:\> $user = read-host 'Enter user name (SAMaccountname)'
Enter user name (SAMaccountname): pgeyer
PS myAD:\> try {
>>     Set-ADUser -identity $user -PasswordNeverExpires $False
>>     Set-ADUser -identity $user -ChangePasswordAtLogon $True
>>     write-host "Attribute updated successfully for user $user" -foregroundcolor yellow
>> } catch {
>>     write-host "Problem in updating attribute for user $user" -foregroundcolor red
>> }
Attribute updated successfully for user pgeyer

```

The following examples are self-explanatory.

```
$identity = 'pgeyer'  
Disable-ADaccount -identity $identity  
Enable-ADaccount -identity $identity  
Unlock-Adaccount -identity $identity  
Remove-ADUser -identity $identity -whatif
```

```
PS myAD:\>  
PS myAD:\> $identity = 'pgeyer'  
PS myAD:\> Disable-ADaccount -identity $identity  
PS myAD:\> Enable-ADaccount -identity $identity  
PS myAD:\> Unlock-Adaccount -identity $identity  
PS myAD:\> Remove-ADUser -identity $identity -whatif  
what if: Performing the operation "Remove" on target "CN=Phil Geyer,OU=An
```

Working with Groups:

Listing groups.

```
Get-AdGroup 'Linux_Fans'  
  
Get-ADGroup -LDAPfilter '(cn=Linux_Fans)' `  
-SearchBase 'ou=AnnArbor,dc=cis161-100,dc=local'
```

```
PS myAD:\>
PS myAD:\> Get-AdGroup 'Linux_Fans'

DistinguishedName : CN=Linux_Fans,OU=AnnArbor,DC=CIS161-100,DC=Local
GroupCategory     : Security
GroupScope        : DomainLocal
Name              : Linux_Fans
ObjectClass       : group
ObjectGUID        : 667aa43a-ffc8-45e3-bbc7-f400057a26be
SamAccountName   : Linux_Fans
SID               : S-1-5-21-2841689656-2999035272-1982814355-1106

PS myAD:\>
PS myAD:\> Get-ADGroup -LDAPfilter '(cn=Linux_Fans)' ` 
>>      -SearchBase 'ou=AnnArbor,dc=cis161-100,dc=local'

DistinguishedName : CN=Linux_Fans,OU=AnnArbor,DC=CIS161-100,DC=Local
GroupCategory     : Security
GroupScope        : DomainLocal
Name              : Linux_Fans
ObjectClass       : group
ObjectGUID        : 667aa43a-ffc8-45e3-bbc7-f400057a26be
SamAccountName   : Linux_Fans
SID               : S-1-5-21-2841689656-2999035272-1982814355-1106
```

The example below searches tree for a group called 'Linux_Fans'. This is not the most efficient method when the Active Directory tree is very large. Since the group object has to be found the a recurse search for the user objects is required.

```
Get-ADGroupMember 'Linux_Fans'
```

```
PS myAD:\>
PS myAD:\> Get-ADGroupMember 'Linux_Fans'

distinguishedName : CN=Jane Doe,OU=AnnArbor,DC=CIS161-100,DC=local
name              : Jane Doe
objectClass        : user
objectGUID         : 6d4c63a8-eace-42bc-ad52-7f5ae650b387
SamAccountName    : jdoe
SID                : S-1-5-21-2841689656-2999035272-1982814355-1107

distinguishedName : CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=local
name              : Phil Geyer
objectClass        : user
objectGUID         : 92892c2a-f87d-47ab-9809-67d08b49a8dc
SamAccountName    : pgeyer
SID                : S-1-5-21-2841689656-2999035272-1982814355-1105
```

This is more efficient since the cmdlet is able to find the group object directly using the distinguished name.

```
Get-ADGroupMember -identity 'cn=Linux_fans,ou=AnnArbor,dc=cis161-100,dc=local'
```

```
PS myAD:\>
PS myAD:\> Get-ADGroupMember -identity 'cn=Linux_Fans,ou=AnnArbor,dc=cis161-100,dc=local'

distinguishedName : CN=Jane Doe,OU=AnnArbor,DC=CIS161-100,DC=local
name              : Jane Doe
objectClass        : user
objectGUID         : 6d4c63a8-eace-42bc-ad52-7f5ae650b387
SamAccountName    : jdoe
SID                : S-1-5-21-2841689656-2999035272-1982814355-1107

distinguishedName : CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=local
name              : Phil Geyer
objectClass        : user
objectGUID         : 92892c2a-f87d-47ab-9809-67d08b49a8dc
SamAccountName    : pgeyer
SID                : S-1-5-21-2841689656-2999035272-1982814355-1105
```

```
clear-host
write-host
$identity = 'pgeyer'
$userobj = get-aduser -Identity $identity
```

```

write-host "Group membership for $($userobj.name)
`($($userobj.SAMAccountName)`)"
Get-ADGroup -filter * | foreach-object {
    $groupname = $_.name
    $userobj = $_ | Get-ADGroupMember | ? { $_.SAMAccountName -eq
$identity }
    if ( $userobj ) {
        write-host "`tGroup $groupname";
    }
}

```

```

1 clear-host
2 write-host
3 $identity = 'pgeyer'
4 $userobj = get-aduser -Identity $identity
5 write-host "Group membership for $($userobj.name) `(`($($userobj.SAMAccountName)`)"
6 Get-ADGroup -filter * | foreach-object {
7     $groupname = $_.name
8     $userobj = $_ | Get-ADGroupMember | ? { $_.SAMAccountName -eq $user }
9     if ( $userobj ) {
10         write-host "`tGroup $groupname";
11     }
12 }
13

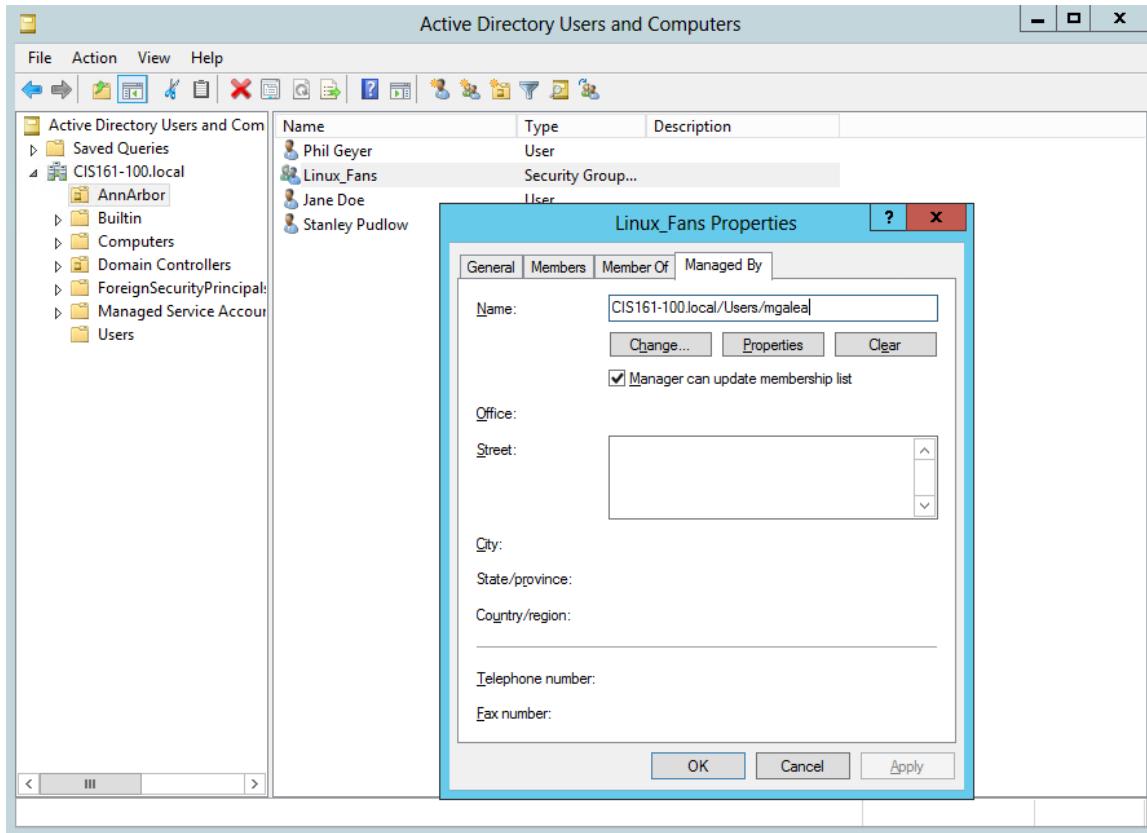
```

```

Group membership for Phil Geyer (pgeyer)
  Group Domain Users
  Group Linux_Fans

```

Adding members to a group requires that the user executing the cmdlet be listed as manager of the group. Use the "Active Directory Users and Computers" console and edit the properties of the group in question as illustrated below.



Once the user is listed as the manager, the `Add-ADGroupMember` cmdlet may be executed against the group. The `-Members` parameter requires a user identity list be it the SAMAccountName or distinguished name.

```
Add-ADGroupMember -identity 'Linux_Fans' -Members jdoe, spudlow
```

```
PS myAD:>
PS myAD:> Add-ADGroupMember -identity 'Linux_Fans' -Members jdoe, spudlow
PS myAD:>
PS myAD:> Get-ADGroupMember -identity 'Linux_Fans'

distinguishedName : CN=Stanley Pudlow,OU=AnnArbor,DC=CIS161-100,DC=local
name              : Stanley Pudlow
objectClass        : user
objectGUID         : 70ddfd3a-3c1e-413e-bb9a-8ff4207dcba2
SamAccountName    : spudlow
SID               : S-1-5-21-2841689656-2999035272-1982814355-1108

distinguishedName : CN=Jane Doe,OU=AnnArbor,DC=CIS161-100,DC=local
name              : Jane Doe
objectClass        : user
objectGUID         : 6d4c63a8-eace-42bc-ad52-7f5ae650b387
SamAccountName    : jdoe
SID               : S-1-5-21-2841689656-2999035272-1982814355-1107

distinguishedName : CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=local
name              : Phil Geyer
objectClass        : user
objectGUID         : 92892c2a-f87d-47ab-9809-67d08b49a8dc
SamAccountName    : pgeyer
SID               : S-1-5-21-2841689656-2999035272-1982814355-1105
```

As indicated earlier to eliminate any ambiguity, a distinguished name should be used.

```
Add-ADGroupMember -identity 'cn=Linux_fans,ou=AnnArbor,dc=cis161-100,dc=local'
-Members jdoe, spudlow
```

Other cmdlets are available to add or remove groups and to set group properties.

```
PS myAD:\>
PS myAD:\> help *ADGroup*
```

Get-ADGroupMember	Cmdlet	activedirectory	...
New-ADGroup	Cmdlet	activedirectory	...
Remove-ADGroup	Cmdlet	activedirectory	...
Remove-ADGroupMember	Cmdlet	activedirectory	...
Set-ADGroup	Cmdlet	activedirectory	...

Group Policy Objects

Monday, April 27, 2015
2:13 PM

<https://technet.microsoft.com/en-us/library/ee461027.aspx>

The get-gpo cmdlet retrieves the group policies for the domain. The -all switch parameter causes the cmdlet to retrieve all group policies.

```
get-gpo -all
```

```
PS myAD:\>
PS myAD:\> get-gpo -all

DisplayName      : Default Domain Policy
DomainName      : CIS161-100.local
Owner           : CIS161-1000\Domain Admins
Id              : 31b2f340-016d-11d2-945f-00c04fb984f9
GpoStatus        : AllSettingsEnabled
Description      :
CreationTime     : 3/30/2015 4:19:49 PM
ModificationTime : 3/30/2015 3:39:06 PM
UserVersion      : AD Version: 0, SysVol Version: 0
ComputerVersion  : AD Version: 3, SysVol Version: 3
WmiFilter        :

DisplayName      : Default Domain Controllers Policy
DomainName      : CIS161-100.local
Owner           : CIS161-1000\Domain Admins
Id              : 6ac1786c-016f-11d2-945f-00c04fb984f9
GpoStatus        : AllSettingsEnabled
Description      :
CreationTime     : 3/30/2015 4:19:49 PM
ModificationTime : 3/30/2015 3:19:48 PM
UserVersion      : AD Version: 0, SysVol Version: 0
ComputerVersion  : AD Version: 1, SysVol Version: 1
WmiFilter        :
```

A specific group policy may be retrieved by specifying the name.

```
get-gpo -name 'Default Domain Policy' | format-list *
```

```
PS myAD:\>
PS myAD:\> get-gpo -name 'Default Domain Policy' | format-list *

Id : 31b2f340-016d-11d2-945f-00c04fb984f9
DisplayName : Default Domain Policy
Path : cn={31B2F340-016D-11D2-945F-00C04FB984F9},cn=pol
Owner : CIS161-1000\Domain Admins
DomainName : CIS161-100.local
CreationTime : 3/30/2015 4:19:49 PM
ModificationTime : 3/30/2015 3:39:06 PM
User : Microsoft.GroupPolicy.UserConfiguration
Computer : Microsoft.GroupPolicy.ComputerConfiguration
GpoStatus : AllSettingsEnabled
WmiFilter :
Description :
```

The get-gpo cmdlet provides the properties of the group policy object. The get-gporeport cmdlet produces a detailed report of the group policy in either xml or html format.

```
get-gporeport -name 'Default Domain Policy'
               -path c:\temp\domain-gpo.html -reporttype html
```

```
PS myAD:\>
PS myAD:\> get-gporeport -name 'Default Domain Policy' ` 
>>           -path c:\temp\domain-gpo.html -reporttype html
>>
PS myAD:\>
PS myAD:\> invoke-item c:\temp\domain-gpo.html
PS myAD:\> -
```

The invoke-item in the above opens domain-gpo.html in a browser window.

The screenshot shows the 'Default Domain Policy' properties window. The 'General' tab is selected. Under 'Details', it lists the following information:

Domain	CIS161-100.local
Owner	CIS161-100\Domain Admins
Created	3/30/2015 4:19:48 PM
Modified	3/30/2015 4:39:06 PM
User Revisions	0 (AD), 0 (SYSVOL)
Computer Revisions	3 (AD), 3 (SYSVOL)
Unique ID	{31B2F340-016D-11D2-945F-00C04FB984F9}
GPO Status	Enabled

The 'Links' section shows a single link to 'CIS161-100' with 'Enforced' status and 'Enabled' link status. The path is 'CIS161-100.local'. Below this, a note states: 'This list only includes links in the domain of the GPO.'

The 'Security Filtering' section shows the settings apply to the following groups, users, and computers:

Name	NT AUTHORITY\Authenticated Users
------	----------------------------------

The 'Delegation' section lists the groups and users with specified permissions:

Name	Allowed Permissions	Inherited
NT AUTHORITY\Authenticated Users	Read (from Security Filtering)	No
NT AUTHORITY\ENTERPRISE DOMAIN CONTROLLERS	Read	No
NT AUTHORITY\SYSTEM	Edit settings, delete, modify security	No

The 'Computer Configuration (Enabled)' section is expanded, showing the following navigation path:

- Policies
- Windows Settings
- Security Settings
- Account Policies/Password Policy
- Policy
- Setting

The Get-GPIinheritance cmdlet shows the inherited group policies for an OU in the Active Directory.

```
Get-GPIinheritance -target 'dc=cis161-100,dc=local'
```

```
PS myAD:\>
PS myAD:\> Get-GPIinheritance -target 'dc=cis161-100,dc=local'

Name          : cis161-100.local
ContainerType : Domain
Path          : dc=cis161-100,dc=local
GpoInheritanceBlocked : No
GpoLinks      : {Default Domain Policy}
InheritedGpoLinks : {Default Domain Policy}
```

```
Get-GPIheritance -target 'ou=AnnArbor,dc=cis161-100,dc=local'
```

```
PS myAD:\>
PS myAD:\> Get-GPIheritance -target 'ou=AnnArbor,dc=cis161-100,dc=local'

Name          : annarbor
ContainerType : OU
Path          : ou=annarbor,dc=cis161-100,dc=local
GpoInheritanceBlocked : No
GpoLinks      : {}
InheritedGpoLinks : {Default Domain Policy}
```

Group policies may be created and linked to Active Directory. The sequence below creates a group policy; assigns it to the AnnArbour OU; and grants edit permission to members of the "Ann Arbor Admins" group located in the AnnArbor OU.

```
# Create the GPO
New-GPO -Name AnnArborGPO -comment "Test GPO to be applied to the
AnnArbor OU"

# Retrieve the GPO and link it the Ann Arbor OU
Get-GPO -Name AnnArborGPO | New-GPlink -target
'ou=AnnArbor,dc=cis161-100,dc=local'

# Give the "Ann Arbor Admins" group the permission to edit the group
policy
Set-GPPermissions -Name AnnArborGPO -permissionlevel gpoedit `

    -TargetName 'Ann Arbor Admins' -TargetType group
```

```

PS myAD:\>
PS myAD:\> # Create the GPO
PS myAD:\> New-GPO -Name AnnArborGPO -comment "Test GPO to be applied to the AnnArbor OU"

DisplayName      : AnnArborGPO
DomainName       : CIS161-100.local
Owner            : CIS161-1000\mgalea
Id               : 9bb7bbd5-d045-4f86-8584-5d43e5aab41e
GpoStatus        : AllSettingsEnabled
Description       : Test GPO to be applied to the AnnArbor OU
CreationTime     : 11/11/2015 8:05:20 PM
ModificationTime : 11/11/2015 8:05:20 PM
UserVersion      : AD Version: 0, SysVol Version: 0
ComputerVersion  : AD Version: 0, SysVol Version: 0
WmiFilter        :

PS myAD:\> Get-GPO -Name AnnArborGPO | New-GPLink -target 'ou=AnnArbor,dc=cis161-100'

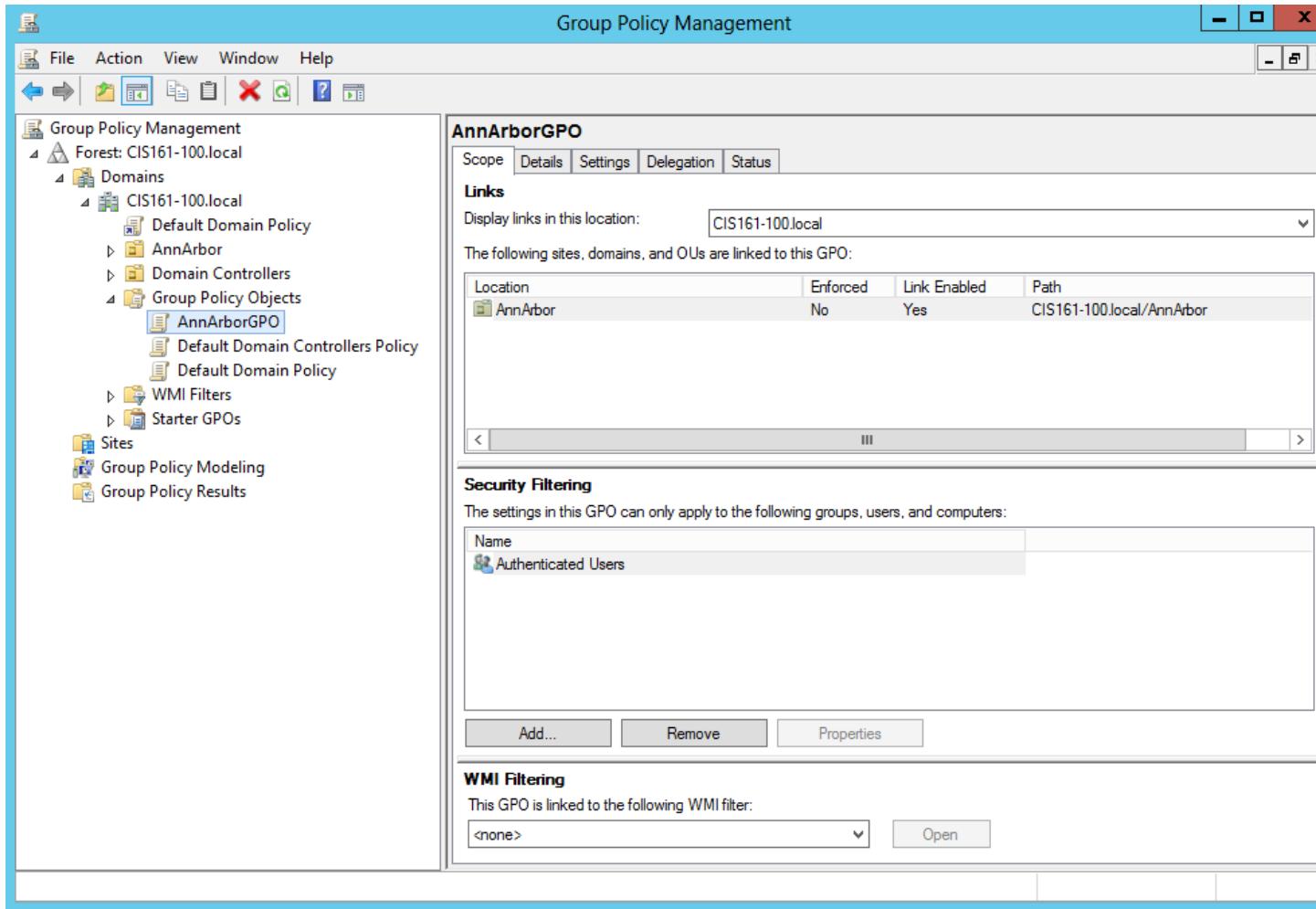
GpoId      : 9bb7bbd5-d045-4f86-8584-5d43e5aab41e
DisplayName: AnnArborGPO
Enabled    : True
Enforced   : False
Target     : OU=AnnArbor,DC=CIS161-100,DC=local
Order      : 1

PS myAD:\> Set-GPPermissions -Name AnnArborGPO -permissionlevel gpoedit
>>           -TargetName 'Ann Arbor Admins' -TargetType group
>>

DisplayName      : AnnArborGPO
DomainName       : CIS161-100.local
Owner            : CIS161-1000\mgalea
Id               : 9bb7bbd5-d045-4f86-8584-5d43e5aab41e
GpoStatus        : AllSettingsEnabled
Description       : Test GPO to be applied to the AnnArbor OU
CreationTime     : 11/11/2015 8:05:20 PM
ModificationTime : 11/11/2015 8:05:20 PM
UserVersion      : AD Version: 0, SysVol Version: 0
ComputerVersion  : AD Version: 0, SysVol Version: 0
WmiFilter        :

```

The Group Policy is now created and may be edit with the Group Policy Management snap-in.



Other GPO cmdlets are available to manage Group Policies.

```
PS myAD:>
PS myAD:> help *-gpo*
```

Cmdlet	Description	...
Get-GPO	GroupPolicy	...
Get-GPReport	GroupPolicy	...
Import-GPO	GroupPolicy	...
New-GPO	GroupPolicy	...
Remove-GPO	GroupPolicy	...
Rename-GPO	GroupPolicy	...
Restore-GPO	GroupPolicy	...

Using .Net

Tuesday, April 23, 2013
8:47 AM

Active Directory cmdlets are preferred particularly when working directly on the Domain Controller. When using the AD cmdlets on a workstation, the Remote Server Administration Tools must be installed. An alternative to installing RSAT is using .Net which doesn't require RSAT. The scripts below may be executed on any version of the Windows Operating System that supports PowerShell.

Querying Active Directory:

```
$ADRoot = New-Object DirectoryServices.DirectoryEntry(  
    "LDAP://10.10.10.100",  
    "CIS161-100.local\administrator","cntpassword")
```

```
PS C:\Users>  
PS C:\Users> $ADRoot = New-Object DirectoryServices.DirectoryEntry(  
    "LDAP://10.10.10.100",  
    "CIS161-100.local\administrator",  
    "cntpassword")  
PS C:\Users>  
PS C:\Users> $adroot  
  
distinguishedName : {DC=CTS161-100,DC=local}  
Path : LDAP://10.10.10.100
```

The \$ADRoot variable provides access to the directory structure. The subordinate containers under the root of the directory tree may be viewed as using the syntax \$ADRoot.children as shown below:

```
PS C:\Users>
PS C:\Users> $ADRoot.psbase.children

distinguishedName : {OU=AnnArbor,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/OU=AnnArbor,DC=CIS161-100,DC=local

distinguishedName : {CN=Builtin,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=Builtin,DC=CIS161-100,DC=local

distinguishedName : {CN=Computers,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=Computers,DC=CIS161-100,DC=local

distinguishedName : {OU=Domain Controllers,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/OU=Domain Controllers,DC=CIS161-100,DC=local

distinguishedName : {CN=ForeignSecurityPrincipals,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=ForeignSecurityPrincipals,DC=CIS161-100,DC=local

distinguishedName : {CN=Infrastructure,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=Infrastructure,DC=CIS161-100,DC=local

distinguishedName : {CN=LostAndFound,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=LostAndFound,DC=CIS161-100,DC=local

distinguishedName : {CN=Managed Service Accounts,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=Managed Service Accounts,DC=CIS161-100,DC=local

distinguishedName : {CN=NTDS Quotas,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=NTDS Quotas,DC=CIS161-100,DC=local

distinguishedName : {CN=Program Data,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=Program Data,DC=CIS161-100,DC=local

distinguishedName : {CN=System,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=System,DC=CIS161-100,DC=local

distinguishedName : {CN=TPM Devices,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=TPM Devices,DC=CIS161-100,DC=local

distinguishedName : {CN=Users,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=Users,DC=CIS161-100,DC=local
```

A connection may also be made to a specific container in the tree. The script below connects to the AnnArbor OU in the root of the AD tree.

```
$ADHost = '10.10.10.100'
$username = 'administrator'
$pwd='cntpASSWORD'
# Branch in the AD tree to which we want to connect
$LDAPconnect = "LDAP://$ADHost/ou=AnnArbor,dc=cis161-100,dc=local"
$ADAnnArbor = New-Object
DirectoryServices.DirectoryEntry($LDAPconnect,$username,$pwd)
```

```

PS C:\Users> $ADHost = '10.10.10.100'
PS C:\Users> $username = 'administrator'
PS C:\Users> $pwd='cntpssword'
PS C:\Users> # Branch in the AD tree to which we want to connect
PS C:\Users> $LDAPConnect = "LDAP://$ADHost/ou=AnnArbor,dc=cis161-100,dc=local"
PS C:\Users> $ADAnnArbor = New-Object DirectoryServices.DirectoryEntry($LDAPConnect, $username, $pwd)
PS C:\Users> $adannarbor

distinguishedName : {OU=AnnArbor,DC=CIS161-100,DC=local}
Path               : LDAP://10.10.10.100/ou=AnnArbor,dc=cis161-100,dc=local

```

Given that the above logon is successful, the variable \$ADAnnArbor is pointing to the AnnArbor OU under cis161-100.local.

```
$users = $ADAnnArbor.children
$users | format-list *
```

```

PS C:\Users>
PS C:\Users> $users = $ADAnnArbor.children
PS C:\Users> $users | format-list *

objectClass          : {top, group}
cn                  : {Ann Arbor Admins}
member              : {CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=local}
distinguishedName   : {CN=Ann Arbor Admins,OU=AnnArbor,DC=CIS161-100,DC=local}
instanceType         : {4}
whenCreated          : {11/12/2015 1:02:29 AM}
whenChanged          : {11/12/2015 1:02:55 AM}
uSNCreated           : {System.__ComObject}
uSNChanged           : {System.__ComObject}
name                : {Ann Arbor Admins}
objectGUID           : {64 135 184 193 22 139 226 65 163 150 131 58 222 131 2}
objectSid             : {1 5 0 0 0 0 5 21 0 0 0 56 190 96 169 136 165 193 17}
sAMAccountName       : {Ann Arbor Admins}
sAMAccountType       : {536870912}
groupType            : {-2147483644}
objectCategory        : {CN=Group,CN=Schema,CN=Configuration,DC=CIS161-100,DC=local}
dsCorePropagationData: {1/1/1601 12:00:00 AM}
nTSecurityDescriptor: {System.__ComObject}
AuthenticationType    : {Secure}
Children              : {}
Guid                 : {4087b8c1168be241a396833ade83f976}
ObjectSecurity        : {System.DirectoryServices.ActiveDirectorySecurity}
NativeGuid            : {4087b8c1168be241a396833ade83f976}
NativeObject           : {System.__ComObject}
Parent                : {LDAP://10.10.10.100/ou=AnnArbor,dc=cis161-100,dc=local}
Password              :
Path                 : {LDAP://10.10.10.100/CN=Ann Arbor Admins,ou=AnnArbor,dc=cis161-100,dc=local}
Properties             : {objectClass, cn, member, distinguishedName...}
SchemaClassName        : {group}
SchemaEntry             : {System.DirectoryServices.DirectoryEntry}
UsePropertyCache       : {True}

```

This script enumerates the users under the AD node pointed to by \$ADAnnArbor which is the AnnAbor OU.

```
write-host "User in AnnArbor OU"
$users = $ADAnnArbor.children | where { $_.ObjectClass -eq 'user' }
foreach ($user in $users) {
    write-host $user.Name $user.SAMAccountName
}
```

```
1  clear-host
2  write-host
3  $ADHost = '10.10.10.100'
4  $username = 'administrator'
5  $pwd='cntpword'
6  # Branch in the AD tree to which we want to connect
7  $LDAPconnect = "LDAP://$ADHost/ou=AnnArbor,dc=cis161-100,dc=local"
8  $ADAnnArbor = New-Object DirectoryServices.DirectoryEntry($LDAPconnect,$username,$pw
9
10 write-host "Use in AnnArbor OU"
11 $users = $ADAnnArbor.children | where { $_.ObjectClass -eq 'user' }
12 foreach ($user in $users) {
13     write-host $user.Name $user.SAMAccountName
14 }
15
```

```
Use in AnnArbor OU
Jane Doe jdoe
Phil Geyer pgeyer
Stanley Pudlow spudlow
```

In a similar manner, groups enumerated.

```
$groups = $ADAnnArbor.children | where { $_.ObjectClass -eq 'group' }
}
$groups |
    format-table @{Expression={$_.Name};Label="Group";width=25},
                @{Expression={$_.member};label="Members";width=50} -
wrap
```

```

PS C:\Users>
PS C:\Users> $groups = $ADAnn Arbor.children | where { $_.ObjectClass -eq 'group' }
PS C:\Users> $groups |
>>>   Format-Table @{Expression={$_.Name};Label="Group";width=25},
>>>           @{Expression={$_.member};label="Members";width=50} -wrap

```

Group	Members
Ann Arbor Admins	CN=Phil Geyer,OU=Ann Arbor,DC=CIS161-100,DC=local {CN=Stanley Pudlow,OU=Ann Arbor,DC=CIS161-100,DC=local,CN=Jane Doe,OU=Ann Arbor,DC=CIS161-100,DC=local,CN=Phil Geyer,OU=Ann Arbor,DC=CIS161-100,DC=local}
Linux_Fans	

The following script takes a different approach. This script enumerates the groups and lists the members of the group

```

$groups = $ADAnn Arbor.children | where { $_.ObjectClass -eq 'group' }
}
foreach ($group in $groups) {
    $GroupName = $group.name
    # Where in the AD tree is the branch for the members of the group
    $GroupPath = $group.Path
    # connect to that branch
    $ADGroupMembers = New-Object DirectoryServices.DirectoryEntry(
        $GroupPath,'administrator','cntpassword')
    write-host 'Group: ' $GroupName
    foreach ($member in $ADGroupMembers.member) {
        write-host `t $member
    }
}

```

```

1  clear-host
2  write-host
3  $ADHost = '10.10.10.100'
4  $username = 'administrator'
5  $pwd='cntpword'
6  # Branch in the AD tree to which we want to connect
7  $LDAPconnect = "LDAP://$ADHost/ou=AnnArbor,dc=cis161-100,dc=local"
8  $ADAnnArbor = New-Object DirectoryServices.DirectoryEntry($LDAPconnect,$username,$pwd
9
10 $groups = $ADAnnArbor.children | where { $_.ObjectClass -eq 'group' }
11 foreach ($group in $groups) {
12     $GroupName = $group.name
13     # Where in the AD tree is the branch for the members of the group
14     $GroupPath = $group.Path
15     # connect to that branch
16     $ADGroupMembers = New-Object DirectoryServices.DirectoryEntry(
17         $GroupPath,'administrator','cntpword')
18     write-host 'Group: ' $GroupName
19     foreach ($member in $ADGroupMembers.member) {
20         write-host `t $member
21     }
22 }
```

Group: Ann Arbor Admins
 CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=local
 Group: Linux_Fans
 CN=Stanley Pudlow,OU=AnnArbor,DC=CIS161-100,DC=local
 CN=Jane Doe,OU=AnnArbor,DC=CIS161-100,DC=local
 CN=Phil Geyer,OU=AnnArbor,DC=CIS161-100,DC=local

Updating Active Directory;

The script below illustrates the steps necessary to create an user object in Active Directory. Note that LDAP attributes for the user object are required. The create method of the OU object is used to create the user object. The put method of the newly created user object is used to update the LDAP attributes. The commit method is required to commit the changes made to the user object. The password is set after the user object is created.

```

$LDAPconnect = "LDAP://10.10.10.100/ou=AnnArbor,dc=cis161-
100,dc=local"
$ADAnnArbor = New-Object DirectoryServices.DirectoryEntry(
    $LDAPconnect,'administrator','cntpword')
```

```

# create the user object
$ADUser = $ADAnn Arbor.Create("User", "cn=Khaled Mansour")
$ADUser.Put('samaccountname','kmansour')
$ADUser.Put('givenname','Khaled')
$ADUser.Put('sn','Mansour')
$ADUser.Put('department','Computer Information Systems')
$ADUser.Put('street','4800 E Huron River Drive')
$ADUser.Put('st','Michigan')
$ADUser.Put('postalcode','48105')
$ADUser.SetInfo() # commit the change

# Set the password. The user object must exist.
$password = 'cntpassword'

$ADUser.Invoke('SetPassword',$password)

```

```

1 $LDAPconnect = "LDAP://10.10.10.100/ou=Ann Arbor,dc=cis161-100,dc=local"
2 $ADAnn Arbor = New-Object DirectoryServices.DirectoryEntry(
3   $LDAPconnect,'administrator','cntpassword')
4
5 # create the user object
6 $ADUser = $ADAnn Arbor.Create("User", "cn=Khaled Mansour")
7 $ADUser.Put('samaccountname','kmansour')
8 $ADUser.Put('givenname','Khaled')
9 $ADUser.Put('sn','Mansour')
10 $ADUser.Put('department','Computer Information Systems')
11 $ADUser.Put('street','4800 E Huron River Drive')
12 $ADUser.Put('st','Michigan')
13 $ADUser.Put('postalcode','48105')
14 $ADUser.SetInfo() # commit the change|
15
16 # Set the password. The user object must exist.
17 $password = 'cntpassword'
18
19 $ADUser.Invoke('SetPassword',$password)

```

```

PS C:\Users>
PS C:\Users> $ADAnn Arbor.children | where-object { $_.sn -eq 'Mansour' }

distinguishedName : {CN=Khaled Mansour,OU=Ann Arbor,DC=CIS161-100,DC=local}
Path              : LDAP://10.10.10.100/CN=Khaled Mansour,ou=Ann Arbor,dc=cis161-100,dc=local

```

Using ADSIEdit (on Server 2012) to inspect the user object properties in AD:

The screenshot shows the ADSI Edit interface. On the left is a tree view of the Active Directory structure under 'Default naming context [CIS]'. In the center, a table lists several users with their names, class (group or user), and distinguished name. The user 'CN=kmansour' is selected. A detailed properties dialog box for 'CN=kmansour Properties' is open in the foreground. This dialog has tabs for 'Attribute Editor' (selected) and 'Security'. Under 'Attributes', there is a table with columns 'Attribute' and 'Value'. The 'userAccountControl' attribute is highlighted with a yellow background, showing its value as '0x222 = (ACCOUNTDISABLE | PASSWD_NEEDED)'. Other attributes listed include title, uid, uidNumber, unicodePwd, unixHomeDirectory, unixUserPassword, url, userCert, userCertificate, userParameters, userPassword, userPKCS12, and userPrincipalName, all with '<not set>' as their value.

Notice, that userAccountControl has a value of 546 (0x222). This indicates that the account is disabled and no password is required. This <http://support.microsoft.com/kb/305144> provides and definition of the userAccountControl codes. The codes are additive so 546 means:

NORMAL_ACCOUNT	512
PASSWD_NOTREQD	32
ACCOUNTDISABLE	2

To enable the account and require a password on the next login, change the userAccountControl to 8389120:

NORMAL_ACCOUNT	512
PASSWD_EXPIRED	8388608

There is no specific value for enabling the account, since 2 is not included in the total the account is enabled.

```
$uac = 0x800200 # hex equivalent of 512 + 8383608 -> 8389120
$ADUser.put('userAccountControl',$uac)
$ADUser.SetInfo()
```

```
PS C:\Users>
PS C:\Users> $ADUser

distinguishedName : {CN=kmansour,OU=AnnArbor,DC=CIS161-100,DC=Local}
Path              : LDAP://10.10.10.100/CN=kmansour,ou=AnnArbor,dc=cis161-100,dc=local

PS C:\Users> $uac = 0x800200 # hex equivalent of 512 + 8383608 -> 8389120
PS C:\Users> $ADUser.put('userAccountControl',$uac)
PS C:\Users> $ADUser.SetInfo()
```

The user object may be deleted using the Delete method of the container object as shown below.

```
$ADAnnArbor.Delete("User", 'cn=kmansour')
```

Using [ADSI]

Tuesday, April 23, 2013
3:52 PM

The [ADSI] accelerator may be used when working directly on the server. The examples below assume that the user is signed into the server as the administrator. In which case, there is no need to authenticate to AD.

To create a user object, we must first connect to the container where the user object is to reside. The script below creates a user object in the OU AnnArbor. The Create method creates a user object in memory. The last step in the script invokes the SetInfo() method which commits the user object in memory to Active Directory. If this step is omitted, the object is not actually created in Active Directory

```
# connect to the container where the user is to be created
$ADAnnArbor = [adsi] 'LDAP://ou=AnnArbor,dc=cis161-100,dc=local'

# create the user object in memory
$ADUser = $ADAnnArbor.Create("User", "cn=Khaled Mansour")
$ADUser.SetInfo() # commit the change
```

After the user object is created, we may then update attributes of the object as shown in the script below. In the case, we must first connect to the AD object as shown in the first line of the script.

```
$ADUser = [adsi] 'LDAP://cn=Khaled Mansour,ou=AnnArbor,dc=cis161-100,dc=local'
$ADUser.samaccountname = 'kmansour'
$ADUser.givenname='Khaled'
$ADUser.sn='Mansour'
$ADUser.department='Computer Information System'
$ADUser.street='4800 E Huron River Drive'
$ADUser.st='Mi'
$ADUser.postalcode='48105'

$ADUser.SetInfo() # commit the change
```

Working with local SAM accounts;

When managing local user accounts and groups, the [ADSI] accelerator may be used. The syntax shown below established the connection to the local repository.

```
$LocalRoot = [ADSI]"WinNT://localhost"
```

Authentication is not an issue since the user running the script is already authenticated to the local computer.

Unlike the connection to Active Directory, the properties and methods of the directoryentry object are hidden. To view them, we need to reference the *PSMemberSet* using *psbase*.

```
PS C:\Users>
PS C:\Users> $LocalRoot = [ADSI]"WinNT://localhost"
PS C:\Users> $localroot.psbase | gm
```

Type Name: System.Management.Automation.PSMemberSet

Name	MemberType	Definition
Disposed	Event	System.EventHandler Disposed(System.Object, void Close())
Close	Method	void CommitChanges()
CommitChanges	Method	adsi CopyTo(adsi newParent), adsi CopyTo(adsi CreateObjRef)
CopyTo	Method	System.Runtime.Remoting.ObjRef CreateObjRef()
CreateObjRef	Method	void DeleteTree()
DeleteTree	Method	void Dispose(), void IDisposable.Dispose()
Dispose	Method	bool Equals(System.Object obj)
Equals	Method	int GetHashCode()
GetHashCode	Method	System.Object GetLifetimeService()
GetLifetimeService	Method	type GetType()
GetType	Method	System.Object InitializeLifetimeService()
InitializeLifetimeService	Method	System.Object Invoke(string methodName, Params)
Invoke	Method	System.Object InvokeGet(string propertyName)
InvokeGet	Method	void InvokeSet(string propertyName, Params)
InvokeSet	Method	void MoveTo(adsi newParent), void MoveTo(adsi NativeObject)
MoveTo	Method	void RefreshCache(), void RefreshCache(string newName)
RefreshCache	Method	string ToString()
Rename	Method	System.DirectoryServices.AuthenticationType AuthenticationType
ToString	Method	System.DirectoryServices.DirectoryEntries Children
AuthenticationType	Property	System.ComponentModel.IContainer Container
Children	Property	guid Guid {get;}
Container	Property	string Name {get;}
Guid	Property	string NativeGuid {get;}
Name	Property	System.Object NativeObject {get;}
NativeGuid	Property	
NativeObject	Property	

To view the objects in the local repository, we reference the child objects as below.

```
PS C:\Users>
PS C:\Users> $localroot.psbase.children

distinguishedName :
Path : WinNT://WORKGROUP/localhost/Administrator

distinguishedName :
Path : WinNT://WORKGROUP/localhost/DefaultAccount

distinguishedName :
Path : WinNT://WORKGROUP/localhost/Guest

distinguishedName :
Path : WinNT://WORKGROUP/localhost/mike

distinguishedName :
Path : WinNT://WORKGROUP/localhost/Access Control Assistant

distinguishedName :
Path : WinNT://WORKGROUP/localhost/Administrators

distinguishedName :
```

To access an object in the local repository, we connect to the object using the path shown above.

```
PS C:\Users>
PS C:\Users> $LocalUser = [ADSI]"WinNT://workgroup/localhost/Guest"
PS C:\Users> $localUser | format-list *

UserFlags : {66147}
MaxStorage : {-1}
PasswordAge : {0}
PasswordExpired : {0}
LoginHours : {255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255}
FullName : {}
Description : {Built-in account for guest access to the computer/}
BadPasswordAttempts : {0}
LastLogin : {6/20/2015 1:25:21 PM}
HomeDirectory : {}
LoginScript : {}
Profile : {}
HomeDirDrive : {}
Parameters : {}
PrimaryGroupID : {513}
Name : {Guest}
MinPasswordLength : {0}
MaxPasswordAge : {600000}
```

The script below creates a local user then updates some properties. Similar to working with Active Directory, the user object created in memory must first be committed to the repository before user object attributes may be modified.

```
$LocalUser = $LocalRoot.Create("User", 'pgeyer')
$LocalUser.SetInfo()

$LocalUser = [ADSI]"WinNT://workgroup/localhost/pgeyer"
$LocalUser.setPassword('hatesWindows@@')
$LocalUser.FullName = 'Phillip Geyer'
$LocalUser.SetInfo()
```

Thursday, December 21, 2017
2:22 PM

Created with Microsoft OneNote 2016.