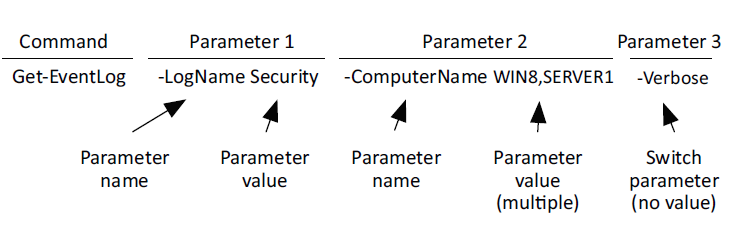
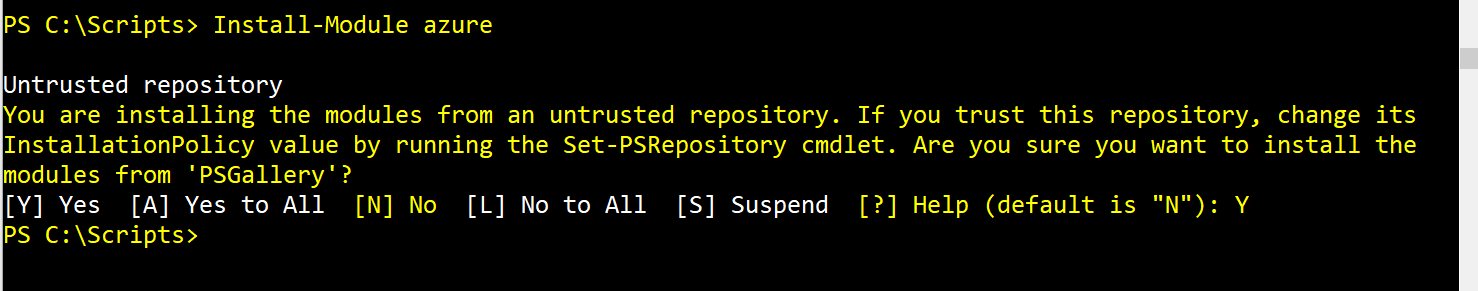
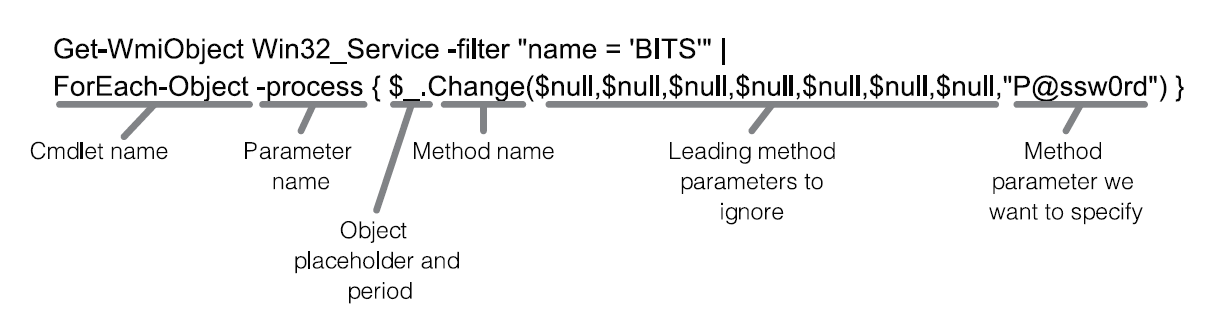
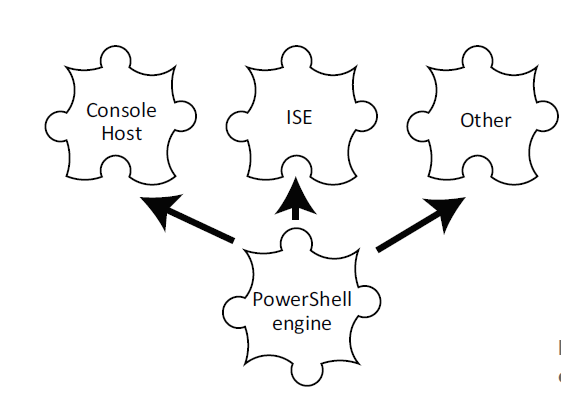
1. Meet PowerShell
   1. Console
      1. Tab
      2. Administrator
   2. ISE – Integrated Scripting Engine
      1. Tab
      2. Administrator
      3. IntelliSense
   3. VSCode
      1. Version
         1. V1.0 folder refers to the language version of the engine.
         2. $PSversiontable
2. Using the help system
   1. Update Help
   2. How to discover commands
   3. Get-help
      1. <http://github.com/PowerShell>
      2. -example
      3. -full
   4. Get-command
   5. Show-command
   6. Get-member
   7. About\_
   8. Optional and mandatory parameters
      1. [[parameter] <type[]>] = optional
      2. [parameter] <type>= mandatory and positional
      3. Parameter <type> = non-positional
3. Running commands
   1. Command structure
      1. 
   2. Cmdlet naming convention
      1. Cmdlet
      2. Function
      3. Workflow
      4. Application
      5. Command
      6. Aliases
   3. Errors
      1. Get help
      2. Dir c:\windows /S
4. Working with providers
   1. Get-help about\_providers
   2. Get-psproviders
      1. ShouldProcess
      2. Filter
      3. Credentials
      4. Transactions
   3. Item – Any Object
5. The pipeline: connecting commands
   1. Connects one command to another
   2. Exporting CSV or XML
   3. Comparing Files
   4. Out-file or Out-printer
   5. Out-Gridview
   6. Convertto-html
6. Objects: data by another name
   1. Get-Process | ConvertTo-HTML | Out-File processes.html
   2. Object—This is what we’ve been calling a table row. It represents a single thing, such as a single process or a single service.
   3. Property—This is what we called a table column. It represents one piece of information about an object, such as a process name, process ID, or service status.
   4. Method—This is what we called an action. A method is related to a single object and makes that object do something—for example, killing a process or starting a service.
   5. Collection—This is the entire set of objects, or what we’ve been calling a table.
   6. get-process | get-member | where {$\_.membertype -eq 'Property'} | group MemberType | sort count -Descending
   7. Why use objects
      1. Tasklist pulls information but you can’t perform any action on it
      2. Powershell pulls information and preform actions on it
      3. PowerShell sc is alias for set-content
      4. Taskill sc preforms actions on services
      5. PowerShell can parse faster (get-process | select processname)
      6. Discover objects with Get-member
         1. \*Propterties member types are out of scope of this class
         2. Actions (stop-process object ; object| stop-process; object.kill()
         3. Sort objects (acending; decending)
         4. Select properties
            1. Select – choose what to see
            2. Where - removes objects from the pipeline
      7. Objects till the end
         1. Get-Process | Sort-Object VM -descending | Out-File c:\procs.txt
         2. Select-object creates a PSCustomObject
7. Logic and flow control
   1. IF/Else - run code blocks if a specified conditional test evaluates to true
      1. if (<test1>) {<statement list 1>}
         1. [elseif (<test2>) {<statement list 2>}]
         2. [else {<statement list 3>}]
   2. Switch - To check multiple conditions
      1. Switch (<test-value>)
         1. { <condition> {<action>}
         2. <condition> {<action>} }
   3. Foreach – iterates through a series of values
      1. foreach ($<item> in $<collection>) {<statement list>}
   4. For - use to create a loop that runs commands in a command block while a specified condition evaluates to true
      1. for (<init>; <condition>; <repeat>) {<statement list>}
   5. Do-Until - the script block runs only while the condition is false
      1. do {<statement list>} until (<condition>)
   6. Do-While - condition is evaluated after the script block has run
      1. do {<statement list>} while (<condition>)
   7. While - runs commands in a command block as long as a conditional test evaluates to true
      1. while (<condition>){<statement list>}
8. Variables: a place to store your stuff
   1. Variable names usually contain letters, numbers, and underscores, and it’s most common for them to begin with a letter or an underscore.
   2. Variable names can contain spaces, but the name must be enclosed in curly braces. For example, ${My Variable} represents a variable named My Variable.
   3. Variables don’t persist between shell sessions.
   4. Variable names can be quite long—long enough that you don’t need to worry about how long. Try to make variable names sensible.
   5. PowerShell users don’t typically use variable name prefixes to indicate what’s stored in the variable.
   6. Assign a variable
      1. $var = 5
   7. Variables inside quotes
      1. Single quotes are a literal string; variable will not expand
      2. Double quotes will execute variables and commands
      3. Demo
         1. $var = 5
         2. $String = ‘What does $var contain?’
         3. $String2 = “What does $var contain?”
            1. Only expands once
         4. $var = 89; $string2
         5. ` (backtick) escapes the variable expansion and command execution
            1. $string3 = "`$var contains $var"
            2. "`$(get-date) is how you execute a command within quotes"
            3. `n is a newline

$objectList = "Object01 `nObject02 `nObject03"

help about\_escape

* 1. Variables with multiple objects
     1. Access by index number
        1. $ListVariables = 'Object1', 'Object2', 'Object3', 'Object4'
        2. $var[0]
        3. $var[-1] last object
        4. $var[-2] 2nd to last objects
        5. $var.length
        6. $var.touppper()
        7. $var.tolower()
        8. $var.replace()
     2. Previous creates new strings not replace current
        1. You have re-declare the variable to change it.
  2. Other Ways to work with variables
     1. $services = Get-Service
     2. $services.Name
     3. Get-Service | ForEach-Object { Write-Output $\_.Name }
     4. Get-Service | Select-Object –ExpandProperty Name
     5. $() – sub expression
        1. $($services[0].name)
  3. Declare Variable type
     1. $number = Read-host “Enter a number: “
        1. 100
     2. $number | get-member
        1. System.String not System.Int32
     3. [int]$number = Read-Host "Enter a number"
     4. Common variable types
        1. [single] and [double]—Single(32)-precision and double(64)-precision floating numbers (numbers with a decimal portion)
        2. [string]—A string of characters
        3. [char]—Exactly one character (as in, [char]$c = 'X')
        4. [xml]—An XML document; whatever string you assign to this will be parsed to make sure it contains valid XML markup (for example, [xml]$doc = Get-Content MyXML.xml)
        5. [adsi]—An Active Directory Service Interfaces (ADSI) query; the shell will execute the query and place the resulting object or objects into the variable (such as [adsi]$user = "WinNT:\\MYDOMAIN\Administrator,user")
        6. [char] A Unicode 16-bit character
        7. [byte] An 8-bit unsigned character
        8. [int] 32-bit signed integer
        9. [long] 64-bit signed integer
        10. [bool] Boolean True/False value
        11. [decimal] A 128-bit decimal value
        12. [DateTime] Date and Time
        13. [array] An array of values
        14. [hashtable] Hashtable object
  4. Variable commands
     1. New-Variable
     2. Set-Variable
     3. Remove-Variable
     4. Get-Variable
     5. Clear-Variable
  5. Variable Best Practice
     1. Keep variable names meaningful but succinct. Whereas $computername is a great variable name because it’s clear and concise, $c is a poor name, because what it contains isn’t clear.
     2. Don’t use spaces in variable names.

1. Adding commands
   1. Just like adding snap-ins into the mmc console
      1. Add; get; remove-snapin
      2. Get-PSSnapin -registered
      3. gcm -pssnapin sqlservercmdletsnapin100
      4. get-psprovider
   2. Import-modules (product specific management shells)
      1. Mini-shells
      2. get-content env:psmodulepath (
         1. used for module Autodiscovery
         2. used for update help to grab the correct update files
      3. get; import-module (install powerCLI)
   3. Modules tend to have a prefix
      1. ADuser; sqlcmd; vm\*
      2. This is to avoid name collisions (last one loaded wins)
   4. Note the differences between PowerShell as a product and all things PowerShell can touch
   5. Profile scripts (
      1. use export-console c:\myshell.psc for snapins
         1. PS shortcut to point to:
         2. %windir%\system32\WindowsPowerShell\v1.0\powershell.exe -noexit -psconsolefile c:\myshell.psc
      2. Profile Script steps to learn how to use them on Windows:
         1. In your Documents folder, create a new folder called Windows PowerShell (no spaces in the folder name; it’s different on non-Windows operating systems, but you can run $profile in the shell to see the correct path).
      3. In the newly created folder, use Notepad to create a file named profile.ps1. When you save the file in Notepad, be sure to enclose the filename in quotation marks (“profile.ps1”). Using quotes prevents Notepad from adding a .txt filename extension. If that .txt extension gets added, this trick won’t work.
      4. In that newly created text file, type your Add-PSSnapin and Import-Module commands, listing one command per line in order to load your preferred snapins and modules.
      5. Back in PowerShell, you need to enable script execution, which is disabled by default. Security consequences may result that we’ll discuss in chapter 17, but for now we’ll assume you’re doing this in a standalone virtual machine, or on a standalone test computer, and that security is less of an issue. In the shell, run Set-ExecutionPolicy RemoteSigned. Note that the command works only if you’re on Windows and run the shell as Administrator. It’s also possible for a Group Policy object (GPO) to override this setting; you’ll get a warning message if that’s the case.
      6. Assuming you haven’t had any errors or warnings up to this point, close and reopen the shell. It automatically loads profile.ps1, executes your commands, and loads your favorite snap-ins and modules for you.
   6. Getting Modules from the internet
      1. 
      2. Uses PowerShellGet
         1. ISEScriptingGeek or EnhancedHTML2
      3. Find-module (<http://powershellgallery.com>)
      4. Register-PSRepository to add the URL of a repository
2. The pipeline, deeper
   1. Accept output from commandA as input for commandB
      1. CommandB has to have property that accepts pipeline input
         1. ByValue
         2. ByPropertyName
         3. Get-help get-service -full
            1. Look for accepted pipeline (ByValue; ByPropertyName)
            2. Name; inputobject; computername
      2. ByValue
         1. Get-content .\servers.txt | Get-service (fail)
         2. Get-process -name note\* | stop-process(win)
         3. Match only one pipeline input ByValue
      3. ByPropertyName
         1. get-service -name s\* | stop-process (fail values don’t match)
         2. get-service -name s\* | gm
         3. get-help stop-process -full
         4. Matches all that accept pipeline input ByPropertyName
      4. Success when creating your own csv
      5. import-csv .\newusers.csv | select-object -property \*, @{name='samAccountName';expression={$\_.login}}, @{label='Name';expression={$\_.login}}, @{n='Department';e={$\_.Dept}}
         1. create a hash table to change values to what you want.
      6. Hash table is like a python dictionary; uses key=value pair @{}
         1. Must start with Name, N, Label or L
         2. Second part must start with Expression or E
            1. Script block {}
      7. Some commands don’t take computer name from pipeline
         1. Get-wmiobject
            1. Get-content computers.txt | get-wmiobject -class win32\_bios (Fail)
         2. Must use parenthesis
            1. Get-wmiobject (get-content computernames.txt)
         3. Can select property name
            1. -property
            2. -expandproptery
            3. Get-Service -computerName (Get-ADComputer -filter \* searchbase "ou=domain controllers,dc=company,dc=pri" | Select-Object -expand name)
3. Formatting—and why it’s done on the right
   1. Formatting done by rules specified for object
      1. .format.ps1xml
   2. Second Formatting rule (looks for user pre-defined rules)
      1. Types.ps1xml
   3. Third formatting rule
      1. Four or less properties; format as table
      2. Five or more properties; format as list
   4. Format-Table
      1. -autosize (adjusts to column width)
         1. Get-Process | select -last 15 | Format-Table
      2. -property (instead of piping to select)
         1. Get-Process | select -last 15 | Format-Table -Property MachineName, ID, ProcessName, responding
      3. -groupBy
         1. Get-Service | Sort-Object Status | Format-Table -groupBy Status
      4. -wrap
         1. Get-Service | Format-Table Name,Status,DisplayName -autoSize -wrap
   5. Format-list
      1. -property; groupBy; others
   6. Format-Wide (wide list)
      1. Grabs “name” property and creates a 2 column list
      2. -column to change the # of columns
      3. -property select other property instead of name property
   7. Custom columns and lists
      1. Create a hash table
         1. Get-Process | Format-Table Name, @{name='VM(MB)';expression={$\_.VM / 1MB -as [int]}} -autosize
      2. FormatStrings - specifies a formatting code, causing the data to be displayed according to the specified format.
      3. Width - specifies the desired column width
      4. Alignment - specifies the desired column width
   8. Other Out-\* Commands
      1. V5.1 (Gridview; file; Default; host; Null; Printer; String)
      2. V7r4 (File; Default; Host; Null; String)
4. Filtering and comparisons
   1. Filtering Left – filtering as far left as possible or as close to data as possible (on remote machine; causing less work for the next commands
      1. Each cmdlet has its own way to filter
      2. Get-service can filter -name only
      3. Get-adcomputer can filter on any attribute
   2. Filtering out of pipeline - Where-object is used when there is no -filter or you cannot filter on the property you want
      1. Uses comparison Operators
         1. -eq (=) = Equal
         2. -ne (<>)= Not Equal
         3. -ge (>=)and -le (<=) = Greater than or equal to; less than or equal to
         4. -gt (>) and -lt (<) = Greater than; less than
         5. -ceq, -cne, -cgt, -clt, -cge, -cle = case sensitive
      2. Can stack comparisons using AND, OR
         1. And – both comparisons have to be $True
         2. OR – one or the other have to be $true
         3. -Not = reverses $true and $false
      3. Other String comparisons
         1. -like; -notlike; -clike; -cnotlike
         2. -match, -notmatch, -cmatch, -cnotmatch
      4. Filtering out of pipeline using the where-object
         1. Get-Service | Where Status -eq 'Running'
            1. Simple; new in v3; good when only comparing 1 object
         2. get-service | where-object {$\_.status -eq 'running' -AND StartType -eq 'Manual'}
            1. Original syntax; used for multi compare
      5. Iterative process
         1. Get processes.
         2. Get rid of everything that’s PowerShell.
         3. Sort the processes by virtual memory.
         4. Keep only the top 10 or bottom 10, depending on how you sort them.
         5. Add up the virtual memory for whatever is left.
         6. Get-Process | Where-Object -filter { $\_.Name -notlike 'powershell\*' } | Sort VM -descending | Select -first 10 | Measure-Object -property VM -sum
5. A practical interlude
   1. Walk through of building a script
6. Remote control: one-to-one, and one-to-many
   1. -computername
      * 1. First iteration of remoting
   2. Remote PowerShell
      * 1. Similar to Telent and SSH
        2. Uses WS-Man protocol (Web Services for Management)
           1. Over http (5985) and https (5986)
        3. WinRM is MS implementation of WS-MAN
           1. Installed and enabled by default on Servers 2012r2 and up
           2. Installed and disabled by default on Win7 and up
        4. XML format of object is sent back to originating machine
           1. Serialization – to XML format
           2. Deserialization – from XML format
           3. Only a snapshot
        5. Must configure WinRM on machines that will receive remote PS connections
           1. Enable-PSRemoting (call Set-WSManQuickConfig; starts and sets WinRM service to autostart, Registers PowerShell as an endpoint; creates needed firewall rules.
           2. Anything adapter set to Public can’t have Windows Firewall exceptions
   3. Enter-pssession
      * 1. Enter-PSSession -computerName Server-R2 (one-to-one)
        2. Type commands directly on connected server (no serialization or deserialization)
        3. Must use real name. By default it will not let you use IP or and other DNS alias
        4. Profile Scripts don’t carry over
        5. Still restricted by execution policy
        6. Exit-PSSession ends the session
           1. Closing terminal ends the session
           2. Avoid remote chains
   4. Invoke-command
      * 1. One-to-many
           1. Invoke-Command -computerName Server-R2,Server-DC4,Server12 -command { Get-EventLog Security -newest 200 | Where { $\_.EventID -eq 1212 }}
        2. PowerShell talks to up to 32 computers at once
        3. Can use -throttle to talk to more than 32
        4. -command is alias for -scriptblock
        5. Can use get-content computers.txt or get-adcomuters for -computername
           1. Demo my personal script
   5. Invoke-Command vs -computername
      1. -computername;
         1. computers are contacted sequentially and could take longer
         2. does not contain a PSComputerName property so results my be hard to separate
         3. connection is not made with WinRM; it is some other underlining remote technology which may not be allowed through the firewall
         4. Processing is done on local computer; so all records are brought across the wire then filtered
         5. Results are live and don’t need to be serialized or deserialized (fully functional Objects)
      2. Invoke-command;
         1. Computers are contacted in parallel; command could run more quickly
         2. Output contains a PSComputerName property so results are more distinguishable
         3. Is used over WinRM; so firewall rules can be enabled
         4. Quiers and filters on remote computer then results are sent back to local machine
         5. Results need to be serialized and deserialized before and after transmitting over the wire (snapshot results) (limited objects)
   6. Local vs. Remote Processing
      1. Invoke-Command -computerName Server-R2, Server-DC4,Server12 -command { Get-EventLog Security -newest 200 | Where { $\_.EventID -eq 1212 }}
         1. Processed remotely
      2. Invoke-Command -computerName Server-R2,Server-DC4,Server12 -command { Get-EventLog Security -newest 200} | Where { $\_.EventID -eq 1212 }
         1. Process localally
      3. Demo
         1. Invoke-Command -computerName Server-R2,Server-DC4,Server12 -command { get-service -name XblGameSave | stop-service }
            1. Remotely
         2. Invoke-Command -computerName Server-R2,Server-DC4,Server12 -command { get-service -name notepad} | stop-process
            1. Locally
      4. Decentralized objects (demo
         1. Get-service | get-member
         2. Invoke-command -computername tests -scriptblock {get-service} | get-member
7. Using Windows Management Instrumentation and CIM
   1. Open Management Instrumentation – OMI on some Linux (PowerShell not supported)
   2. Windows contains tens of thousands of Management Information (WMI Explorer)
   3. Orgainized into Namespaces
      1. Root/CIMv2 – OS and hardware information
      2. Root/MicrosoftDNS – DNS Server Information
      3. Root/securityCenter – Firewall, antivirus, and antispyware information
   4. Name space maybe available but not exist root/CIMv2 -> Win32\_LogicalDisk
   5. Instance is a real thing represented
      1. If you only have 1 bios, you will have 1 Win32\_bios
      2. If you have 100 services; you will have 100 Win32\_service
   6. CIM cmdlets are wrappers for WMI commands
   7. Demo WMI Explorer
   8. Search PowerShell for WMI object
      1. C:\> Get-WmiObject -Namespace root\CIMv2 -list | >> where name -like '\*dis\*'
         1. CIM\_ Class are often base classes and access directly
            1. Communicates over RPC – If firewall supports stateful inspection
         2. Win32\_ are Windows specific
            1. Communicates over WS-MAN (WinRM)
   9. Old cmdlets (get-wimobject and invoke-wimMethod)
   10. New cmdlets (Get-CIMinstance and Invoke-CIMMethod)
   11. Get-wmiobject -namespace <namespace> -class <class>
       1. Class parameter is positional; no need to specify it if it is listed first
       2. Namespace parameter is not needed if in default namespace (Root\CIMv2)
   12. Demo;
       1. Get-WmiObject win32\_operatingsystem | gm
       2. Has -filter parameter (See book for usage)
       3. Remoting is done with -computer
          1. Each system is contacted sequentially, produce an error and move on or time out if not available
       4. Can be piped to OUT- ConvertTo- export-
   13. Get-CIMInstance
       1. Similar to get-wmiobject
       2. -classname instead of -class
       3. -list not available; must user -namespace instead
       4. -credental not available; must use invoke-command
8. Multitasking with background jobs
   1. Jobs are background jobs
   2. Synchronous – run in foreground (1 command at a time)
      1. Can respond to input requests
      2. Can see error messages
      3. See results promptly
   3. Asynchronous – run as background jobs (multiple at a time)
      1. Input requests stop the job from running
      2. Must develop error handling
   4. Start-job -scriptblock
      1. Local job has support -computername
      2. Requires remote powershell scripting
   5. WMI as a Job
      1. Used -asjob parameter
      2. Creates a child job for every computer in the list
      3. Uses normal WMI communications
      4. Get-ciminstance requires start-job or invoke-command with get-ciminstance in scriptblock
   6. Remoting as a job
      1. Used invoke-command -asjob
      2. Requires PSv2 or higher with remoting enabled
      3. Has -jobname parameter
   7. Receive-child jobs
      1. By name or job ID
      2. Get-jobs | receive-jobs
      3. Receive-jobs clears them out of cache and cannot be retrieved a second time; must use -keep or out-cliXML
      4. Results are decentralized; can be piped into export-xml, sort-object, format-list, convertto-html; out-file
      5. Hasmore data property
         1. Will be false if there is no data in memory (if you viewed and did not use the -keep parameter)
         2. Will be true if there is data in memory
   8. Commands for jobs
      1. Remove-job – deletes job and any cached results in memory
      2. Stop-job – terminates the command; but you can still retrieve results that are in memory
      3. Wait-job – useful in scripts; script will start a job and needs to wait on results from job to continue.
   9. Scheduled jobs
      1. Different from scheduled tasks
      2. Introduced in v3
      3. New-jobtrigger
      4. New-ScheduledTaskOption
      5. Register-ScheduledJob
         1. Register-ScheduledJob -Name DailyProcList -ScriptBlock {Get-Process } -Trigger (New-JobTrigger -Daily -At 2am) -ScheduledJobOption (New-ScheduledJobOption -WakeToRun -RunElevated)
9. Working with many objects, one at a time
   1. Batching Cmdlets
      1. Some cmdlets can batch (stop-service)
      2. Some object methods cannot and need to be piped to invoke-\*method
         1. Get-CimInstance -classname win32\_networkadapterconfiguration -filter "description like '%intel%'" | Invoke-CimMethod -methodname EnableDHCPinvoke-wmimthod -name enabledhcp
      3. 
      4. 
10. Security alert!
    1. PowerShell does give any additional permissions
       1. If you cant do it in the GUI then you cant do it in a script
       2. Does not defend against malware
       3. “Even though apiece of malware might use PowerShell to do harm, that doesn’t make that malware PowerShell’s problem”
    2. Execution Policy – stops users from being tricked into running scripts
       1. Changed by:
          1. Set-executionpolicy
          2. GPO
          3. Powershell.exe – executionpolicy (overwrites any local or group policy)
       2. Restricted – default, scripts are not run
       3. All-signed - PowerShell will execute any script that has been digitally signed by using a code-signing certificate issued by a trusted certification authority (CA)
       4. Remote-signed – (Microsoft Recommended) PowerShell will execute any local script, and will execute remote scripts if they’ve been digitally signed by using a code-signing certificate issued by a trusted CA.
       5. Unrestricted - All scripts will run
       6. Bypass - This setting bypasses the configured execution policy and should be used only when the hosting application is providing its own layer of script security.
    3. Digital Signatures
       1. Set-AuthenticodeSignature "C:\Scripts\Publish\PowerShell\Files\Backup-Files.ps1" (get-childitem Cert:\CurrentUser\My -CodeSigningcert)
    4. Double click opens script in notepad for editing
    5. Must use relative or absolute path to execute .ps1
       1. Protects from command hijacking
    6. PowerShell v5 Security enhancements
       1. <https://blogs.msdn.microsoft.com/daviddasneves/2017/05/25/powershell-security-at-enterprise-customers/>
       2. <https://www.blackhillsinfosec.com/powershell-logging-blue-team/>
       3. <https://www.stigviewer.com/stig/windows_10/2017-02-21/finding/V-68819>
       4. Module logging - Event ID 4103
          1. Logs PowerShell pipeline execution details during execution including variable initialization, and command invocation
             1. Able to record some de-obfuscated scripts, and also some output data
             2. Available in v3
          2. Get-EventLog -LogName \* | where {$\_.InstanceId -eq "4103"}
       5. Script Block Logging -Event ID 4104
          1. Logs and records all blocks of PowerShell code as they are executing
             1. Captures all de-obfuscated code
             2. Available in v5
          2. HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging
          3. Create Registry key
             1. New-Item HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging -Force
             2. New-ItemProperty HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging -Name "EnableScriptBlockLogging" -PropertyType "DWORD" -Value 1
       6. System-wide transcription
          1. New-ItemProperty HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ Transcription
             1. EnableTranscripting, 1
             2. IncludeInvocationHeader,1
             3. OutputDirectory, [Path]
       7. Language Modes
          1. Disable-WindowsOptionalFeature -Online -FeatureName MicrosoftWindowsPowerShellV2
          2. $ExecutionContext.SessionState.LanguageMode
          3. FULL LANGUAGE
             1. The FullLanguage mode permits all language elements in the session.
          4. RESTRICTED LANGUAGE
             1. In RestrictedLanguage mode, users may run commands (cmdlets, functions, CIM commands, and workflows) but are not permitted to use script blocks.
          5. NO LANGUAGE
             1. NoLanguage mode means no script text of any form is permitted.
          6. CONSTRAINED LANGUAGE
             1. The ConstrainedLanguage mode permits all cmdlets and all PowerShell language elements, but it limits permitted types
11. Input and output
    1. Prompting for, and displaying, information
       1. 
    2. Read-Host
       1. A colon is added to the end of the prompt.
       2. Whatever the user types is returned as the result of the command (technically, it’s placed into the pipeline).
       3. Use a graphical box instead of console readline
          1. <https://docs.microsoft.com/en-us/dotnet/framework/additional-apis/index>
          2. <https://docs.microsoft.com/en-us/powershell/scripting/samples/creating-a-custom-input-box?view=powershell-5.1>
          3. [void][System.Reflection.Assembly]::LoadWithPartialName('Microsoft.VisualBasic')
             1. [void] - part is converting the result of the command into the void data type.

an object that does not have a value of any **type**

data type is a special type that means “throw the result away.”

Another way to do the same thing would be to pipe the result to Out-Null.

* + - * 1. [System.Reflection.Assembly] – represents our application

enclosed the type name in square brackets, as if we were declaring a variable to be of that type

we’re using two colons to access a static method of the type

Static methods exist without us having to create an instance of the type.

* + - * 1. LoadWithPartialName ()

static method we’re using

accepts the name of the framework component we want to load.

* + - 1. $computername = [Microsoft.VisualBasic.Interaction]::InputBox('Enter a computer name','Computer Name','localhost')
         1. [Microsoft.VisualBasic.Interaction]

loaded into memory with the previous command

* + - * 1. 'Enter a computer name'

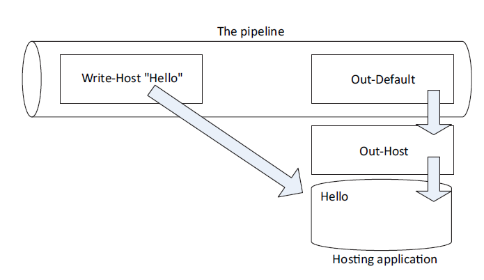
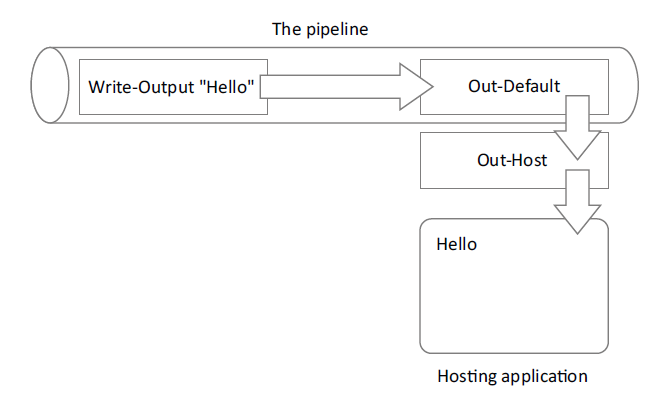
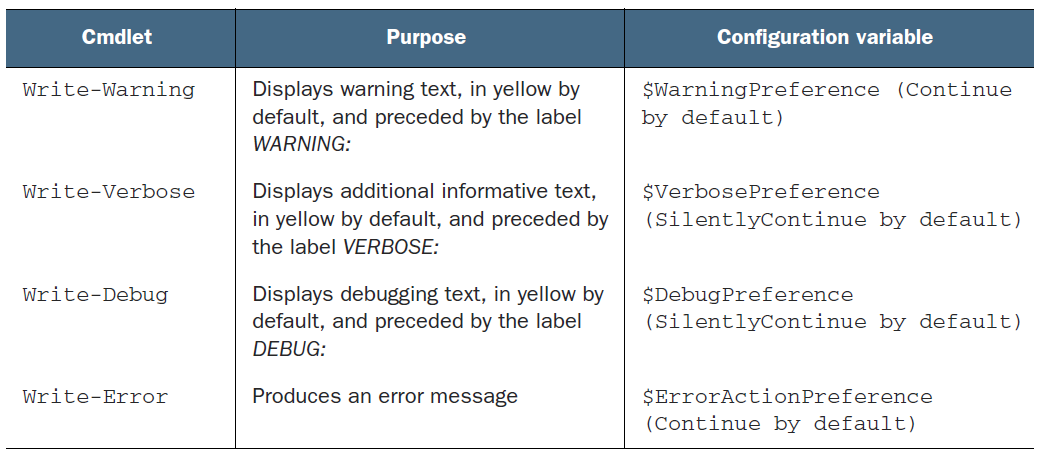
The first parameter is the text for your prompt.

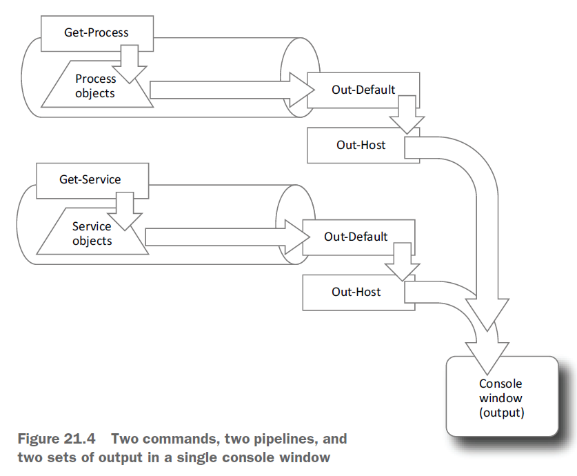
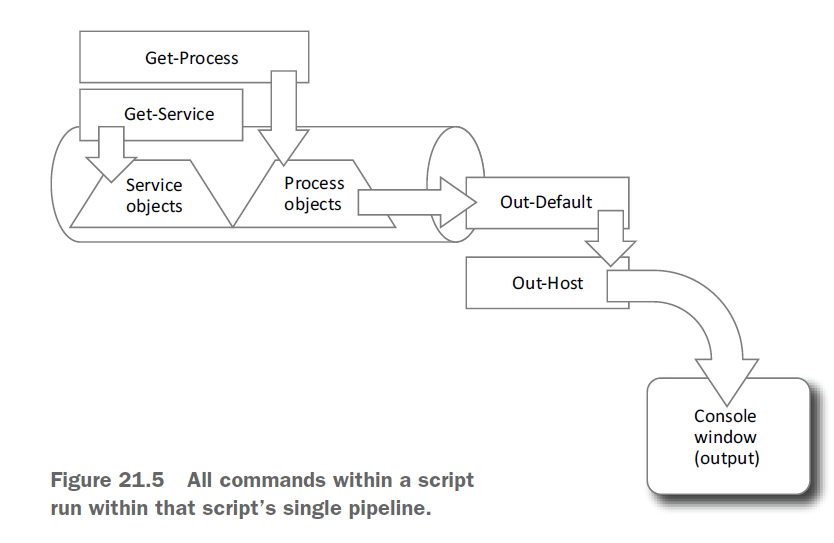
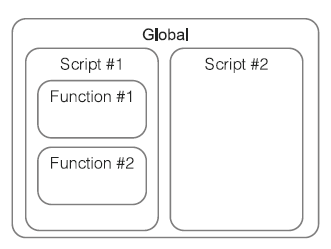
* + - * 1. 'Computer Name'

The second parameter is the title for the prompt’s dialog box.

* + - * 1. 'localhost'

The default value that you want prefilled in the input box.

* 1. Write-Host
     1. 
     2. write-host "COLORFUL!" -fore yellow -back magenta
     3. Anything written to the screen (-host) cannot be captured.
        1. i.e unattended script or remote commands (invoke)
     4. Best practice is to use Write-Verbose
        1. Used for “warm and fuzzy messages”
           1. Connecting to xyz server
           2. Testing connecton
  2. Write-Output
     1. Can send objects into the pipeline
     2. Write-output Hello
        1. Hello gets sent to pipeline to write-default
        2. Write-output Hello | out-default | write-host
     3. 
        1. write-output "Hello" | where-object { $\_.length -gt 10 }
        2. write-output "Hello" | where-object { $\_.length -gt 10 } | out-default
        3. write-output "Hello" | where-object { $\_.length -gt 10 } | out-default | write-host
        4. write-output "Hello" | where-object { $\_.length -gt 10 } | write-host
  3. Other Output
     1. 
     2. Write-Warning
     3. Write-Verbose
     4. Write-Debug
     5. Write-Error
        1. it writes an error to PowerShell’s error stream
     6. Write-Information (v5)
        1. Write-host is a wrapper
        2. May need -informationaction continue
           1. <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_commonparameters?view=powershell-6>
     7. Write-Progress
        1. can display progress bars

1. Sessions: remote control with less work
   1. Creating and using reusable sessions
      1. New-PSSession
         1. Can be used to connect to several machines and stored as a variable
         2. $sessions = New-PSSession -ComputerName localhost, DESKTOP-52INA1A -Credential desktop-52ina1a\honey
            1. Disconnect-PSSession
            2. Connect-PSSession
            3. Remove-PSSession
   2. Using sessions with Enter-PSSession
      * 1. Enter-PSSession -Session $sessions [0]
           1. enter-pssession -session ($sessions |where { $\_.computername -eq ‘localhost' })
           2. enter-pssession -session (get-pssession -computer localhost)(Demo Did not work)
        2. $s\_server1,$s\_server2 = new-pssession -computer localhost, DESKTOP-52INA1A
           1. Optional (one command with 2 individual sessions)
   3. Using sessions with Invoke-Command
      1. invoke-command -command { get-wmiobject -class win32\_process } -session $session
      2. Did not use Get-WMIObject -computername
         1. Invoke-command process on distant machine
         2. Invoke-command works in parallel, -computername processes sequentially
         3. New CIM cmdlets don’t have -computername parameter
      3. invoke-command -command { get-wmiobject -class win32\_process } session (get-pssession –comp server1,server2,server3)
         1. invoke-command -command { get-wmiobject -class win32\_process } -session $session | Select-Object ProcessName, PSComputerName, Path | Group-Object ProcessName | Sort-Object Count -Descending | Format-Table -AutoSize
   4. Implicit remoting: importing a session
      1. Review my Remove-UserScript
      2. $session = new-pssession -comp server-r2
         1. Establish a remote connection to server with ADTools installed
      3. invoke-command -command { import-module activedirectory } session $session
         1. Tell the remote computer to load the AD module
      4. import-pssession -session $session -module activedirectory -prefix rem
         1. import the AD PowerShell Module and prefix the commands with rem
            1. helps with tracking the commands and avoids command conflict
      5. PowerShell creates a temporary local module with shortcuts to the commands on the remote server
      6. Results brought back through the session are decentralized and do not have methods
   5. Using disconnected sessions
      1. Disconnect-pssession
         1. Disconnects from the session, but leaves the connection
         2. Same domain admin can see the connection of a different computer
            1. Admin creates session on Comp1 to Comp2, then disconnects session, logs into Comp3 and checks sessions on Comp2, sees his disconnected session
      2. WSMan: drive manages remote sessions
         1. Get-ChildItem WSMan:\localhost\Shell
            1. IdleTimeout - specifies the amount of time a session can be idle before it’s shut down automatically
            2. MaxConcurrentUsers - specifies the number of users who can have a session open at once
            3. MaxShellRunTime - determines the maximum amount of time a session can be open.
            4. MaxShellsPerUser - sets a limit on the number of sessions a single user can have open at once
         2. Get-ChildItem WSMan:\localhost\Service\
            1. MaxConnections - sets the upper limit on incoming connections to the entire remoting infrastructure.
2. You call this scripting?
   1. Not programming, more like batch files
      1. Can run command sequentially like cmd.exe and batch scripting
      2. ISE makes going from IDE to Script easy
   2. Making Commands repeatable
      1. Demo any script
   3. Parameterizing commands
      1. Add Parameter ()
      2. Separate parameters by a comma
   4. Documenting your script
      1. comment-based help
         1. .SYNOPSIS
         2. .DESCRIPTION
         3. .PARAMETER
         4. .EXAMPLE
   5. One Script, One Pipeline
      1. 
      2. 
   6. A quick look at scope
      1. 
      2. Shell – Global Scope
      3. Script - Script Scope
      4. Functions – Private Scope
      5. Scope.ps1
         1. $x = 10
         2. Write $x
3. Improving your parameterized script
   1. Use get-diskinventory.ps1 and swap format-table with select-object
      1. The point is that outputting objects (which Select-Object does), as opposed to formatted displays, makes our script more flexible in the long run.
      2. Add [CmdletBinding()] to make it an advance script
      3. [Parameter(Mandatory=$True,HelpMessage="Enter a computer name to query")
      4. [Mandatory], [String], [Int], [alias(‘hostname’)]
   2. Validating parameter input
      1. [ValidateSet(2,3)]
      2. help about\_functions\_advanced\_parameters
4. Advanced remoting configuration
   1. Uses custom endpoint configurations (reference my JEA Talk)
   2. Enabling multihop remoting
      1. Second Hop Problem
      2. Enable-WSManCredSSP -Role Client -DelegateComputer $computer (2nd Hop Computer)
      3. Enable-WSManCredSSP -Role Server (ran on middle man computer
   3. Digging deeper into remoting authentication
      1. PowerShell remoting employs mutual authentication
         1. Mostly take care of in a domain environment
            1. The name must resolve to an IP address.
            2. The name must match the computer’s name in the directory.
      2. Mutual authentication via SSL
         1. you need to obtain an SSL digital certificate for the destination machine
         2. you need to create an HTTPS listener
         3. <https://leanpub.com/secretsofpowershellremoting>
      3. TrustedHosts
         1. Shuts off Mutual Authentication
         2. The TrustedHosts item can contain a comma-separated list of computer names, IP addresses, and fully-qualified domain names.
         3. Wildcards are permitted.
         4. Get-Item wsman:\localhost\Client\TrustedHosts
         5. Set-Item wsman:localhost\client\trustedhosts -Value \*
5. Using regular expressions to parse text files
   1. Used with -match and -cmatch (case sensitive)
   2. get-eventlog -LogName security | where { $\_.eventid -eq 4624 } | select -ExpandProperty message | select-string -pattern "WIN[\W\w]+TM[234][0-9]\$"
6. Additional random tips, tricks, and techniques
   1. Operators: -as, -is, -replace, -join, -split, -in, -contains
      1. as - operator produces a new object in an attempt to convert an existing object into a different type
         1. 1000 / 3 -as [int]
      2. Is - It’s designed to return True or False if an object is of a particular type or not
         1. 123.45 -is [int]
         2. "SERVER-R2" -is [string]
         3. $True -is [bool]
         4. (Get-Date) -is [datetime]
      3. Replace - operator is designed to locate all occurrences of one string within another and replace those occurrences with a third string (linux sed command)
         1. "192.168.34.12" -replace "34","15"
      4. join and -split operators are designed to convert arrays to delimited lists, and vice versa (linux Cut, and Awk commands)
         1. $array = "one","two","three","four","five"
         2. $array -join "|"
         3. $string = $array -join "|"
      5. split - It takes a delimited string and makes an array from it
         1. $array = (gc computers.tdf) -split "`t"
      6. Contains – operator is used to test whether a given object exists within a collection
         1. $collection = 'abc','def','ghi','jkl'
         2. $collection -contains 'abc'
      7. Like - operator is designed for wildcard string comparisons
         1. 'this' -contains '\*his\*'
   2. String Manipulation
      1. "Hello" | gm
      2. IndexOf() tells you the location of a given character within the string:
         1. "SERVER-R2".IndexOf("-")
      3. ToLower() and ToUpper() convert the case of a string
         1. $computername = "SERVER17"
         2. $computername.tolower()
      4. Trim() removes whitespace from both ends of a string;
         1. $username = " Don "
         2. $username.Trim()
      5. TrimStart() and TrimEnd() remove whitespace from the beginning or end of a string, respectively
   3. Date manipulation
      1. get-date | gm
         1. $90daysago = $today.adddays(-90)
7. Using someone else’s script
8. Never the end
   1. Further Exploration
      1. PowerShell’s simplified scripting language
      2. Scope
      3. Functions, and the ability to build multiple tools into a single script file
      4. Error handling
      5. Writing help
      6. Debugging
      7. Custom formatting views
      8. Custom type extensions
      9. Script and manifest modules
      10. Using databases
      11. Workflows
      12. Pipeline troubleshooting
      13. Complex object hierarchies
      14. Globalization and localization
      15. Proxy functions
      16. Constrained remoting and delegated administration
      17. Using .NET
9. PowerShell cheat sheet

Random Notes

The LocalAccountTokenFilterPolicy entry disables user account control (UAC) remote restrictions for all users of all affected computers

New-ItemProperty -Name LocalAccountTokenFilterPolicy `

-Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System `

-PropertyType DWord -Value 1