

Windows PowerShell

Introduction

Name

Company


Title, function, job responsibility

Programming, scripting, networking, database experience

Windows PowerShell experience

- What have you heard about PowerShell?
- What do you think about it?

Your expectations from the course



Agenda

What is PowerShell

PowerShell Pipeline

Variables and Data Types, Scope and Collection

Security

Remote Management

Script Flow Control Statements

Functions, Filters and Modules

Error Handling

Scripts

Administrative Uses



What is PowerShell

Windows PowerShell is an interactive object-oriented command environment with scripting language features that utilizes small programs called cmdlets to simplify configuration, administration, and management of heterogeneous environments in both standalone and networked typologies by utilizing standards-based remoting protocols



Introducing PowerShell

Windows PowerShell is both interactive and script

- Big blue Command Prompt window
- Really complicated bunch of code

Object Oriented Programming language

- Output is always a .NET Object

Framework based on .NET

Console and ISE, PowerShell.exe and Powershell_ISE.exe

Why do we need PowerShell

Improved Efficiency + Cost Saving

Standardization, Consistency and Manageability

Re-Usability

Faster

Prevent Human Error

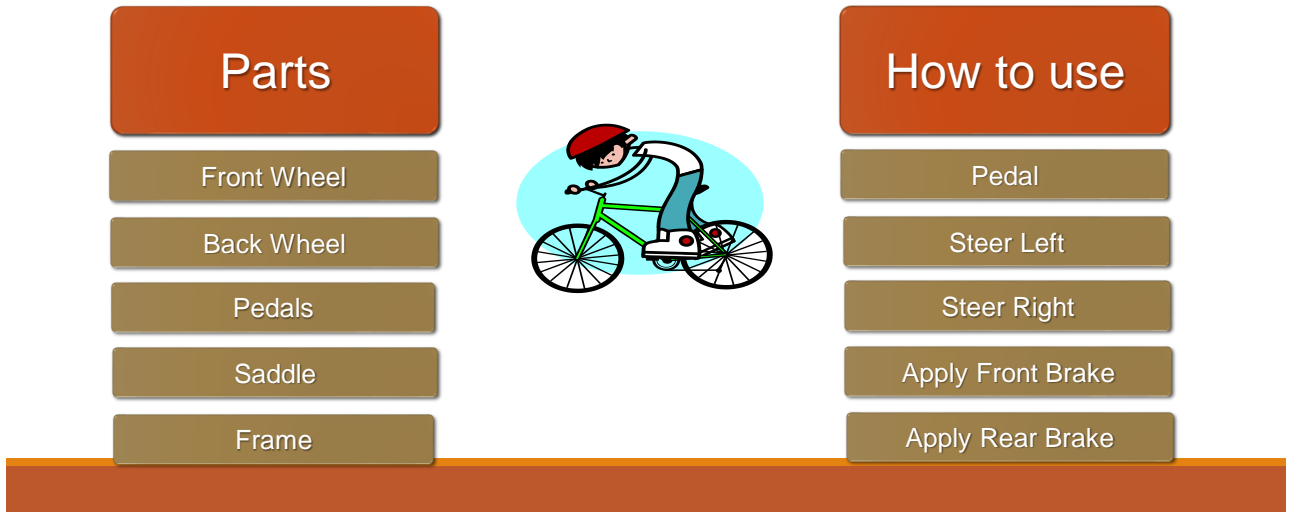
Not Dependent on User Availability

We need to work smarter and not, necessarily HARDER

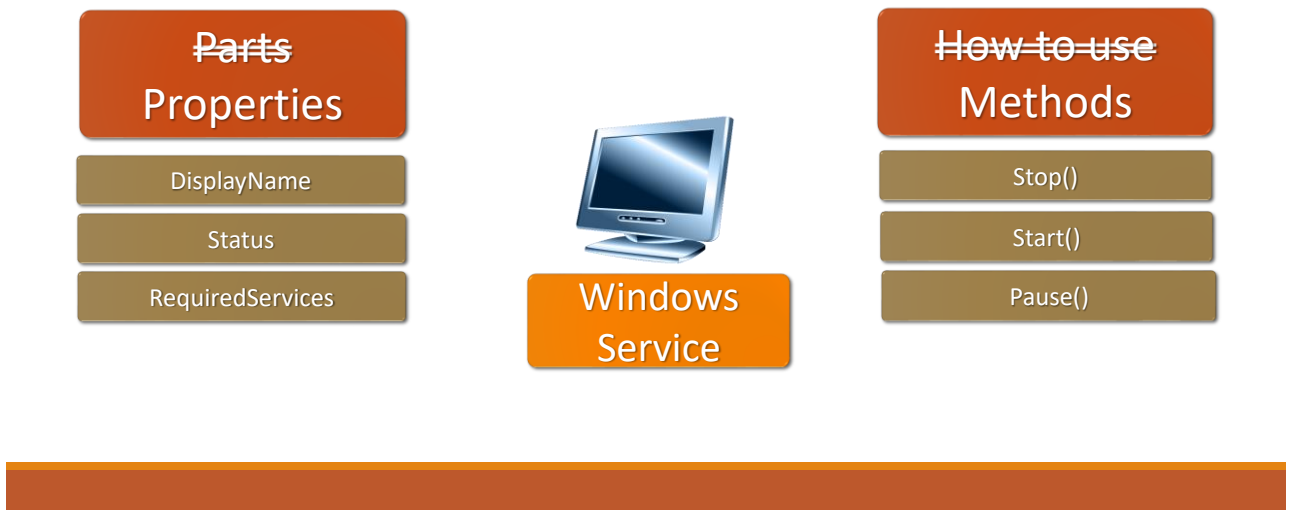
Brief history

Code Name: Monad	1.0	2.0	3.0	4.0
2005	2006	2008	2012	2013
	130 cmdlets	230 cmdlets Backward- Compatible Integrated Shell Environment (ISE) Remoting	>2,300 cmdlets Backward- Compatible WinPE Web Access Enhanced ISE Workflow	>2,300 cmdlets Backward- Compatible Desired State Configuration (DSC)

Objects “An object is a collection of parts and how to use them”

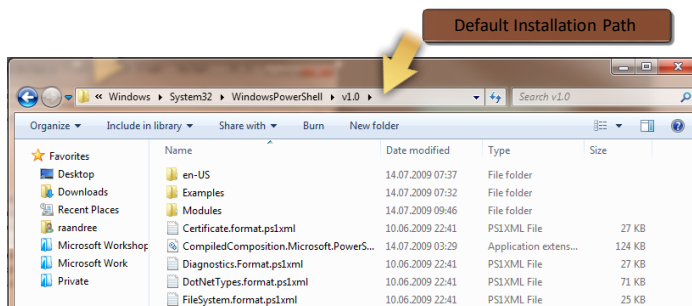


Objects



Where PowerShell lives

- Even PowerShell V3 is in the v1.0 folder
- This is about the same having a System32 folder in a 64 bit Windows OS installation
- Cmdlet Get-Host or \$PSVersionTable prints out information about PowerShell Version



Things that were not possible before

- The PowerShell makes it easy to solve very complex questions with just one line of code like filtering processes...

```
Get-Process
Get-Process | Sort-Object -Property Handles
Get-Process | Sort-Object -Property Handles -Descending
              | Select-Object -First 10
Get-Process | Group-Object Company |
              Sort-Object -Property Name
```

Things that were not possible before

...or finding duplicate files in a folder tree by the attributes Name, Length and LastWriteTime

```
dir -Recurse C:\training |  
  Group-Object -Property Name,Length,LastWriteTime |  
  Where-Object { $_.Count -gt 1 } |  
  ForEach-Object { $_.Group } |  
  Format-Table -Property Name, Length,  
    LastWriteTime, FullName
```

Commands

Cmdlet, Functions, Scripts, and native Windows commands

Cmdlets are 'native' PowerShell commands

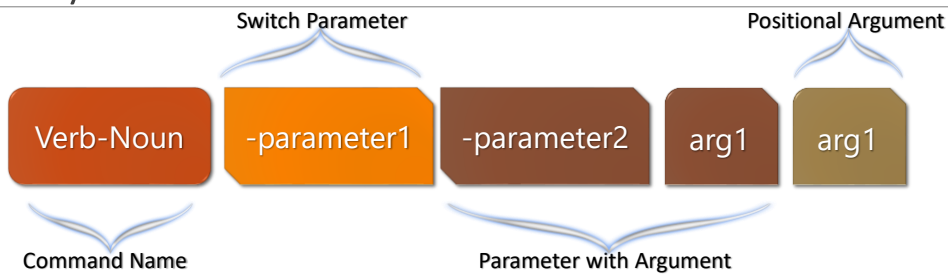
Each cmdlet performs a specific, typically small, task

Cmdlets are written in a .NET language like C# or VB.net

Cmdlet is made of Verb-Noun

```
Get-Help  
Stop-Service Restart-  
Computer Get-  
Command Get-User  
Get-Help  
Get-EventLog  
Get-Process  
Add-Content  
Rename-Item Copy-  
content
```

The syntax to use a cmdlet



```
Get-Process outlook
Get-Process -Name outlook
Stop-Process -Name outlook -Force
```

Cmdlet Syntax

Syntax Definition

```
<Command-Name> -<Required Parameter Name> <Required Parameter Value>
[-<Optional Parameter Name> <Optional Parameter Value>]
[-<Optional Switch Parameters>]
[-<Optional Parameter Name>] <Required Parameter Value>
<Multiple Parameter Value>[]
```

Syntax Sample

```
PS C:\> Get-Command -Name Add-Computer -Syntax
```

```
Add-Computer [-DomainName] <string> -Credential <pscredential> [-ComputerName
<string[]>] [-LocalCredential
<pscredential>] [-UnjoinDomainCredential <pscredential>] [-OUPath <string>] [-
Server <string>] [-Unsecure] [-Options
<JoinOptions>] [-Restart] [-PassThru] [-NewName <string>] [-Force] [-WhatIf] [-
Confirm] [-CommonParameters]
```


Cmdlet Syntax - Command Name

Syntax Definition

```
<Command-Name> -<Required Parameter Name> <Required Parameter Value>
[-<Optional Parameter Name> <Optional Parameter Value>]
[-<Optional Switch Parameters>]
[-<Optional Parameter Name>] <Required Parameter Value>
<Multiple Parameter Values>[]
```

Syntax Sample

```
PS C:\> Get-Command -Name Add-Computer -Syntax
```

```
Add-Computer [-DomainName] <string> -Credential <pscredential> [-ComputerName
<string[]>] [-LocalCredential
<pscredential>] [-UnjoinDomainCredential <pscredential>] [-OUPath <string>] [-
Server <string>] [-Unsecure] [-Options
<JoinOptions>] [-Restart] [-PassThru] [-NewName <string>] [-Force] [-WhatIf] [-
Confirm] [<CommonParameters>]
```

17

Cmdlet Syntax - Required Parameter

Syntax Definition

```
<Command-Name> -Required Parameter Name <Required Parameter Value>
[-<Optional Parameter Name> <Optional Parameter Value>]
[-<Optional Switch Parameters>]
[-<Optional Parameter Name>] <Required Parameter Value>
<Multiple Parameter Values>[]
```

Syntax Sample

```
PS C:\> Get-Command -Name Add-Computer -Syntax
```

```
Add-Computer [-DomainName] <string> -Credential <pscredential> [-ComputerName
<string[]>] [-LocalCredential
<pscredential>] [-UnjoinDomainCredential <pscredential>] [-OUPath <string>] [-
Server <string>] [-Unsecure] [-Options
<JoinOptions>] [-Restart] [-PassThru] [-NewName <string>] [-Force] [-WhatIf] [-
Confirm] [<CommonParameters>]
```

18

Cmdlet Syntax - Optional Parameter and Value

Syntax Definition

```
<Command-Name> -<Required Parameter Name> <Required Parameter Value>
[-<Optional Parameter Name> <Optional Parameter Value>]
[-<Optional Switch Parameters>]
[-<Optional Parameter Name>] <Required Parameter Value>
<Multiple Parameter Values>[]
```

Syntax Sample

```
PS C:\> Get-Command -Name Add-Computer -Syntax
```

```
Add-Computer [-DomainName] <string> -Credential <pscredential> [-ComputerName
<string[]>] [-LocalCredential
<pscredential>] [-UnjoinDomainCredential <pscredential>] [-OUPath <string>] [-
Server <string>] [-Unsecure] [-Options
<JoinOptions>] [-Restart] [-PassThru] [-NewName <string>] [-Force] [-WhatIf] [-
Confirm] [<CommonParameters>]
```

19

Cmdlet Syntax - Switch Parameter

Syntax Definition

```
<Command-Name> -<Required Parameter Name> <Required Parameter Value>
[-<Optional Parameter Name> <Optional Parameter Value>]
[-<Optional Switch Parameters>]
[-<Optional Parameter Name>] <Required Parameter Value>
<Multiple Parameter Values>[]
```

Syntax Sample

```
PS C:\> Get-Command -Name Add-Computer -Syntax
```

```
Add-Computer [-DomainName] <string> -Credential <pscredential> [-ComputerName
<string[]>] [-LocalCredential
<pscredential>] [-UnjoinDomainCredential <pscredential>] [-OUPath <string>] [-
Server <string>] [-Unsecure] [-Options
<JoinOptions>] [-Restart] [-PassThru] [-NewName <string>] [-Force] [-WhatIf] [-
Confirm] [<CommonParameters>]
```

20

Cmdlet Syntax - Optional Parameter, Required Value

Syntax Definition

```
<Command-Name> -<Required Parameter Name> <Required Parameter Value>
[-<Optional Parameter Name> <Optional Parameter Value>]
[-<Optional Switch Parameters>]
[-<Optional Parameter Name>] <Required Parameter Value>
<Multiple Parameter Values>[]
```

Syntax Sample

```
PS C:\> Get-Command -Name Add-Computer -Syntax
```

```
Add-Computer [-DomainName] <string> -Credential <pscredential> [-ComputerName
<string[]>] [-LocalCredential
<pscredential>] [-UnjoinDomainCredential <pscredential>] [-OUPath <string>] [-
Server <string>] [-Unsecure] [-Options
<JoinOptions>] [-Restart] [-PassThru] [-NewName <string>] [-Force] [-WhatIf] [-
Confirm] [<CommonParameters>]
```

21

Cmdlet Syntax - Multiple Parameter Values

Syntax Definition

```
<Command-Name> -<Required Parameter Name> <Required Parameter Value>
[-<Optional Parameter Name> <Optional Parameter Value>]
[-<Optional Switch Parameters>]
[-<Optional Parameter Name>] <Required Parameter Value>
<Multiple Parameter Values>[]
```

Syntax Sample

```
PS C:\> Get-Command -Name Add-Computer -Syntax
```

```
Add-Computer [-DomainName] <string> -Credential <pscredential> [-ComputerName
<string[]>] [-LocalCredential
<pscredential>] [-UnjoinDomainCredential <pscredential>] [-OUPath <string>] [-
Server <string>] [-Unsecure] [-Options
<JoinOptions>] [-Restart] [-PassThru] [-NewName <string>] [-Force] [-WhatIf] [-
Confirm] [<CommonParameters>]
```

22

Cmdlet Syntax

```
PS C:\> Get-Command -Name Stop-Process -Syntax
```

```
Stop-Process [-Id] <int[]> [-PassThru] [-Force] [-WhatIf] [-Confirm]
[<CommonParameters>]
```

```
Stop-Process -Name <string[]> [-PassThru] [-Force] [-WhatIf] [-Confirm]
[<CommonParameters>]
```

```
Stop-Process [-InputObject] <Process[]> [-PassThru] [-Force] [-WhatIf]
[-Confirm] [<CommonParameters>]
```

23

Syntax Legend

<verb-noun>	Command name
-<parameter>	Required parameter name
<value>	Required parameter value
[-<> <>]	Optional parameter and Optional value
[-<>] <>	Optional parameter and Required value
<value[]>	Multiple parameter values

24

Profiles

Six different Profiles

Current User, Current Host – console	\$Home\[My]Documents\WindowsPowerShell\Profile.ps1
Current User, All Hosts	\$Home\[My]Documents\Profile.ps1
All Users, Current Host – console	\$PsHome\Microsoft.PowerShell_profile.ps1
All Users, All Hosts	\$PsHome\Profile.ps1
Current user, Current Host – ISE	\$Home\[My]Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1
All users, Current Host – ISE	\$PsHome\Microsoft.PowerShellISE_profile.ps1

Normally if we talk about PowerShell profile, we refer to *current user, current host* profile

A PowerShell script, having a special name, and it resides in a special place

Must enable the Script Execution policy



Profiles - Labs

Find the profile path

Create CurrentUserAllHosts profile, if it doesn't exist

Add any command to it

Relaunch PowerShell and see if profile gets executed

What are Aliases?

- Aliases are short, easy-to-remember terms that can be used to refer to cmdlets or command elements
- Alternate name for cmdlet, as function, script, or executable file
- Get-ChildItem
 - dir
 - ls
 - gci
- Get-Content
 - type
 - gc
 - cat

Aliases

Built-in Aliases and User-defined Aliases

- Get-Alias
- New-Alias
- Set-Alias
- Export-Alias
- Import-Alias

Cannot assign alias to command and its parameters

- Create Function for that command with its parameters and create alias for function



What are Aliases? Labs

What all are the cmdlets to manage aliases?

Create a new alias for the cmdlet "Get-Help" named "gh"

Get a list of all aliases targeting the cmdlet "Get-ChildItem"

Accessing properties or methods

In this demonstration, you will see how to access members of objects using the dot operator

```
PS C:> Get-Date
Monday, April 06, 2009 1:12:00 AM

PS C:> Get-Date | Get-Member

PS C:> Get-Date.Month
The term 'Get-Date.Month' is not recognized as a
cmdlet, function, operable program, or script file.

PS C:>(Get-Date).Month
4

PS C:> (Get-Date).Ticks
633745773524574159

PS C:\> (Get-Date).AddDays(-7)
Monday, March 30, 2009 1:16:46 AM
```

Accessing properties or methods

In this demonstration, you will see how to access members of objects using the dot operator

```
PS C:> $ts = New-TimeSpan -Seconds 324562
PS C:> $ts
Days           : 3
Hours          : 18
Minutes        : 9
Seconds        : 22
Milliseconds   : 0
...
PS C:> $ts.Days -> 3
PS C:> $ts.Seconds -> 22
PS C:> $ts.Subtract((New-TimeSpan -Seconds 236045))
PS C:> $ts.Negate()
```



Date and Time Labs

When does the date calculation starts? 1601? 1980?

Try to work with the ticks property and one of the Add methods to find out "day 0". What date is tick 1 or when does the .net calendar starts?

Get-Date, AddTicks method or use the New-Object cmdlet with the DateTime object

Create a new time span of 600.000 seconds and store it in some variable.

Subtract that time span of the current time

Get-Date, Subtract Method or subtract operator (-), New-TimeSpan

Pipeline

Unix shells initiated pipeline, Cmd.exe copied it, and PowerShell takes it to next level

Output of one command is input for the next one

- **Get-Process | Select Name, Handles | Sort Handles**
- **Get-Childitem *.txt | Where-Object { \$_.LastWriteTime.Year -lt (Get-Date).Year } | Remove-Item**
 - Pipes Object and does not Pipes text
 - A file is an object, and the date of a file is also an object

Pipeline is always used

Objects are converted to text at the end of Pipeline

Displaying particular properties

Two pipeline mode - Sequential (slow) mode and Streaming (quick) mode

Demo 5

PowerShell Help

Update-Help - updates help from Microsoft or from local share

Should be part of Local Administrator group

PowerShell must be run with elevated privileges

Use -Force, if want to update help more than once in 24hrs.

- **Save-Help -DestinationPath [\\servername\resources\PSv3Help](#)**
- **Update-Help -SourcePath [\\servername\resources\PSv3Help](#)**

Get-help, Help or Man

- **Help -Category Cmdlet -Name *service***
- **Get-help Update-Help, Get-Help Get-Service**
- **-Detailed, -Full, and -Examples**

Get-help Get-Service -ShowWindow



First Few Commands - Labs

Get-Help -Name Get-Process

Get-ExecutionPolicy

Set-ExecutionPolicy Unrestricted

Get-Service

ConvertTo-HTML

- **Get-Service | ConvertTo-HTML -Property Name, Status > C:\services.htm**

Export-CSV

- **Get-Service | Export-CSV c:\service.csv**

Select-Object

- **Get-Service | Select-Object Name, Status | Export-CSV c:\service.csv**

Get-EventLog -Logname "Application"

Get-Process

Stop-Process

- **Stop-Process -Name notepad**
- **Stop-Process -ID 2668**

Get-Command

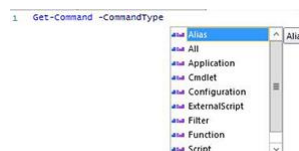
Obtain listing of Cmdlets, Functions, Aliases, Scripts, Workflows, Filters and Applications

Obtain a listing of all the properties of a cmdlet

Listing of cmdlets by filtering it on Noun and Verb

Searches executables too in all folders stored in Path environment variable

- **Get-command -verb get**
- **Get-command -noun service**
- **Get-Command -CommandType cmdlet**
- **Get-Command -name G***
- **Get-Command -Module ActiveDirectory**
 - **\$Command = Get-Command -Name Get-Process**
 - **\$Command.Parameters**
 - **\$Command.Parameters['Name']**
 - **\$Command.Parameters['Name'].Attributes**
- **Get-Command -ParameterName Computername**



Get-Member

What is actually in object and how can you see what else might be lying around?

Returns member types such as properties “.” and methods “ () ”

- `$Directory = Get-ChildItem -Directory`
- `$Directory | Get-Member`
- `$Directory | Get-Member -MemberType Method`

To show members added to object by types.ps1xml file or Add-Member cmdlet

- `$Directory | Get-Member -View Extended`

To view all the members, not shown in default view

- `$Directory | Get-Member -Force`

Get-Service | Get-Member, returns objects of type “System.ServiceProcess.ServiceController”

- Find cmdlets looking to accept 'above' object from the pipeline
- **Get-Command -ParameterType System.ServiceProcess.ServiceController**
- **Get-Help -Name Stop-Service -Full**
- It has a parameter called InputObject that accepts one or more ServiceControllers as input
- InputObject parameter accepts pipeline input

The screenshot shows a PowerShell session in a window titled "Select Windows PowerShell". The prompt is `PS C:\Users\All Users> dir`, and the output is a table of files and directories in `C:\Users\All Users`. Red arrows point from text boxes to specific parts of the output:

- A box says "Get-member (shortcut gm) helps you to find what else output you view given command" with an arrow pointing to the `dir` command.
- A box says "Each Row as object of Class" with an arrow pointing to a row in the `dir` output.
- A box says "Mode Column" with an arrow pointing to the "Mode" column header in the `dir` output.
- A box says "LastWriteTime" with an arrow pointing to the "LastWriteTime" column header in the `dir` output.
- A box says "Name Column" with an arrow pointing to the "Name" column header in the `dir` output.
- A box says "Properties as column (you can say it as Class property)" with an arrow pointing to the "Name" column header in the `Get-Member` output.

The second command is `PS C:\Users\All Users> dir | Get-Member`, which outputs a table of members for the `System.IO.DirectoryInfo` object. The columns are "Name", "MemberType", and "Definition". Red arrows point from text boxes to specific parts of this output:

- A box says "Properties as column (you can say it as Class property)" with an arrow pointing to the "Name" column header.
- A box says "LastWriteTime" with an arrow pointing to the "LastWriteTime" property in the "Definition" column.
- A box says "Mode Column" with an arrow pointing to the "Mode" property in the "Definition" column.

Command Precedence

Rules followed to execute commands when session has commands having same name

If path to a command is specified then command at the location is executed

If path not specified then this order is followed

1. Alias
2. Function
3. Cmdlet
4. Native Windows commands

If session contains items of same type having same name then item that was added to session most recently, is run first

Items can be replaced or hidden by items with same name, like Cmdlets or Variables

- Microsoft.PowerShell.Utility\Get-Date
- MapFunctions\New-Map

Variables

Store information to utilize later on or store result of a command or script

Variables can contain text strings, integers, and even objects

Represented by text strings that begin with \$

Assignment operator "=" sets a variable to a specified value

Cast notation or Strongly Typing Variables, using []

Variable Exists - Test-Path the "drive" called Variable:

Write-Protecting Variables - ReadOnly and Constant

Variable Scope and scope modifier

Interpolation – Single vs Double quotes

- How will you print "The value of the \$psHOME variable is C:\WINDOWS\system32\WindowsPowerShell\v1.0."

Arrays and HashTables

User Created, Automatic and Preference Variables

Automatic variables are pre-defined e.g. \$_, \$Args, \$Error, \$Home, \$PSHome etc.

Environment variables store information about OS environment, they persist and can be modified e.g. WINDIR, PATH

- Child sessions inherit the values
- Virtual Directory - **env**:

Clear-Variable

- Get-Variable
- New-Variable
- Remove-Variable
- Set-Variable

Reserved Words

- Break, continue, do, else, elseif, filter, foreach, function, if, in, return, switch, until, where, while

Data types

PowerShell automatically assigns and converts data to its correct data type

At times, you might want to control this process

Numerical values are not enclosed in quotes

Working with Strings, Numbers, Arrays, Hashtables

Casting - process of changing variable's data type from one value to another

Scope

PowerShell console is the basic scope (global scope)

Each script launched from the console creates its own scope (script scope)

Functions create their own scope

Functions defined inside of other functions create additional sub-scopes

Parent scope cannot access variables defined in a child scope

Child scope can read variables defined in parent but can modify them only using special syntax

Modifying parent scope variable without special syntax, creates a new variable of same name within child scope (Demo)

Scope identifier/modifiers (Demo)

- \$global: works with objects in the global scope
- \$script: works with objects in the parent script scope
- \$local: works with objects in the local scope
- \$private: works with objects in a private scope

Dot Sourcing - forces function/script to run, not in its own scope but rather *right within script/console scope* (Demo)

Scope

Set-Variable-Variable can be read and modified in entire script (but not in parent scope)

- Set-Variable -Name myVariable -Option AllScope

Alternatively, New-Variable cmdlet can be used to create a variable with AllScope property

- New-Variable -Name myVariable -Option AllScope -Value "Available in all child scopes"

Can be referred by number, current scope is referred as zero and its ancestors are referenced by increasing integers

Not just variables get sets in the scope, it's true for aliases, drives, and functions

Get-Help about_scopes

Arrays

Data structures designed to store collections of indexed items

Create an array of a specific size and type by using the New-Object cmdlet

Can access a specific element of an array i.e. 0,1,...,-1,-2

If result of a command is more than one, PowerShell always returns an array

Array sub-expression operator creates array, even if it contains 0 or 1 object “ @() ”

Polymorphic Arrays

Accessing array elements

1. `foreach ($element in $a) {$element}`
2. `for ($i = 0; $i -le ($a.length - 1); $i += 2) {$a[$i]}`
3. `$i=0`
`while($i -lt 4) {$a[$i]; $i++}`

ArrayList - makes it easy to add, remove, insert or even sort array contents

HashTables

Hash tables store "key-value pairs"

The “key” and “value” entries can be any data type and length

Create a new hash table using @{}

Values separated by semi-colons

Hash tables may be used when you want to return text results into Objects

PowerShell Operators

= Assigns a value to a variable

+ or += Addition

- or -= Subtraction

* or *= Multiplication

/ or /= Division

% or %= Modulus (retrieves the remainder of a division operation)



Supplying Parameters for cmdlets

-Whatif

-Confirm

-Verbose (-Verbose:\$true)

-ErrorAction (Continue, Stop, Silently-Continue, and Inquire)

-ErrorVariable

-OutVariable

-OutBuffer



Remoting

Collections

Comparison Operators

Script Flow Control Operators

Security

Remoting – Requirements

PowerShell V2 or later

.NET 2.0 or later

Windows Remote Management 2.0 or later

Current user must be a member of the Administrators group on the remote computer

Or able to provide credentials of an administrator

To enable remoting on client versions of Windows, current Windows network location must be Domain or Private ("Home" or "Work")

Run as Administrator

Remoting

Similar to Telnet/SSH for accessing remote terminals on other operating systems

As HTTP/HTTPS is used, access across Firewalls is easy

Big Deal? WMI with VBScript, PSEXEC, ComputerName parameter

- No consistency between tools
- Across Firewalls
- Tools at times work differently depending on if it is run locally or remotely

Built on Microsoft's implementation of the Web Services for Management (WSMan) and uses WinRM to manage communication and authentication

Massive performance benefits

Imagine running "Get-Service" against 100s of computer one by one and running the same command on each of 100s of computer and sent back the output

Enabling PowerShell Remoting

Currently, remoting is supported on Windows Vista with SP1 or later, Windows 7, Windows Server 2008, and Windows Server 2008 R2 and all the later versions

Locked down by default

Open a PowerShell window as Administrator

Enable-PSRemoting -Force

If in a Workgroup then each computer needs to be configured

Another way to use PowerShell on a remote machine is by using Invoke-Command

- `Invoke-Command - ComputerName <Remotecomputer> -ScriptBlock { <command> } - credential <user>`
- Remote Session - `Enter-PSSession -ComputerName <Remotecomputer> -Credential <user>`

Troubleshooting Remoting

Working with remote computers in other domains

- To enable authentication, add remote computer to list of trusted hosts for local computer in WinRM.
- `winrm s winrm/config/client '@{TrustedHosts="<RemoteComputer>"}`

Working with computers in workgroups

- `Set-Item wsman:\localhost\client\trustedhosts *`
- `Restart-Service WinRM`

Verify that the service on the remote host is running

- `winrm quickconfig`

Test the connection

- `Test-WSMan <RemoteComputer>`

Avoid remote chaining

Demo Remoting

Comparison Operators

`-eq, -ne, -ge, -gt, -lt, -le, -like, -notlike, -match, -notmatch, -contains, -notcontains, -is, -isnot`

- `-contains` is designed to work on arrays, not strings.
- `-match` looks for a match inside a string and supports regexes
- `-like` looks for a match inside a string and supports wildcards

Comparison Operations Examples

Get-childitem | where {\$_.Name -eq "PowerShell notes"} All files where the name of the file is equal to "PowerShell notes"

Get-childitem | where {\$_.Name -ne "PowerShell notes"} All files where the name is not equal to "PowerShell notes"

Get-childitem | where {\$_.Length -gt 1MB} All files where the size is greater than 1MB

Get-childitem | where {\$_.Length -ge 1MB} All files where the size is equal to or greater than 1MB

Gt-childitem | where {\$_.Length -lt 1MB} All files where the size is less than 1MB

Get-childitem | where {\$_.Length -le 1MB} All files where the size is equal to or less than 1MB

Get-childitem | where {\$_.Name -like "PowerShell*"} All files where the name begins with "PowerShell"

Get-childitem | where {\$_.Name -match "Power"} All files where the name contains "Power"

"Microsoft", "Windows", "PowerShell" -contains "Power" Looks if the string "Power" is included in the provided values and will answer "True" to that question

Get-childitem -include *.doc | where {\$_ -replace ".doc", ".docx"} All *.doc files and changes their extension from doc to docx

If-Elseif-Else

<pre>If (\$a -gt 10) { "\$a is larger than 10" } Elseif (\$a -eq 10) { "\$a is exactly 10" } Else { "\$a is less than 10" }</pre>	<pre>Clear-Host \$Name = "Print Spooler" \$Service = Get-Service -display \$Name - ErrorAction SilentlyContinue If (-Not \$Service) {\$Name + " is not installed on this computer."} Else {\$Name + " is installed." \$Name + "'s status is: " + \$Service.Status }</pre>
---	---

If-Elseif-Else

```
$i = 2
if (($i -le 2) -and ($i -gt 2)) {Write-Host "A simple if
program."}
elseif ($i -le 2) {Write-Host "A simple if...else program."}

$filecheck = "D:\Games\solitaire.exe"
$filepresent = Test-Path $filecheck
If ($filepresent -eq $True) {Write-Host "Solitaire is
present"}
else {Write-Host "File isn't present"}
```

Loops

Repeat same set of commands, set number of times

More than one loop technique can be used effectively

Loop condition is usually enclosed in parentheses and body in braces

While ForEach-Object gets input from pipeline, Foreach statement iterates over a collection of objects

- *Foreach* blocks PowerShell until all results are available; for complex commands that can take a very long time
- *Use Foreach* when the results to be evaluated are already completely available

Do and While

- Loop you want to iterate until the condition is true

Loops

Foreach-Object (%) and Foreach

- While ForEach-Object gets input from pipeline, Foreach statement iterates over a collection of objects
- *Foreach* blocks PowerShell until all results are available; for complex commands that can take a very long time
- *Use Foreach* when the results to be evaluated are already completely available

Do and While

- Loop you want to iterate until the condition is true

Loops

For

- When it's known exactly how often you want to iterate a particular code segment
- **Initialization:** Is evaluated when the loop begins
- **Continuation criteria:** Evaluated before every iteration. If \$true, the loop will iterate
- **Increment:** Is likewise re-evaluated with every looping

Exiting Loops Early: Break and Continue

Nested Loops

```
While (! ($file.EndOfStream)) {  
    $file.ReadLine ()  
}  
$file.Close
```

```
Do {  
    $input = Read-Host "Your homepage"  
} While (!(($input -like "www.*.*"))  
$random = New-Object system.random  
For ($i=0; $i -lt 7; $i++) {  
    $random.next(1, 49)  
}
```

Out Cmdlets

Sends command output to a specified device

Name	Description
Out-Default	Sends output to default formatter and to default output cmdlet (Out-Host)
Out-File	Sends output to a file Append switch parameter Encoding parameter allows control of the character encoding
Out-GridView	Sends output to an interactive table in a separate GUI
Out-Host	Default Sends output to PowerShell host Paging switch parameter displays one page at a time
Out-Null	Deletes output instead of sending it down the pipeline
Out-Printer	Sends output to a printer
Out-String	Sends objects to the host as a series of strings

Security

Script Execution

Signing Scripts

Requesting Credentials and Using Secure Strings

Securing Remote Sessions

Script execution

Restricted

- Scripts cannot be run
- PowerShell Interactive only
- Default Setting

Remote Signed

- Runs all local scripts
- Scripts downloaded from IE, Outlook Express and Messenger must be signed by a trusted source

All Signed

- Runs a script only if signed
- Signature must be trusted on local machine

Unrestricted

- All scripts from all sources can be run without signing
- Prompts for scripts coming from the internet

Bypass

- Nothing is blocked and there are no warnings or prompts

Undefined

- Removes the currently assigned execution policy from the current scope

Signing scripts

Adding digital signature to script is signing it with a code signing certificate

- Created by a certificate authority (such as Verisign etc.)
- Created by a user (called a self-signed certificate)

Deciding factor will be Cost, Administrative overhead, Speed and Convenience

For a self-signed certificate, the **makecert.exe** program is required

It is available as part of the .NET Framework SDK or Windows Platform SDK



Signing Scripts, steps outlined

Add Certificate Snap-in to MMC

Setting up a Local Certificate Authority

Create a personal certificate from above CA

Verify the certificate

Sign the script

Test the execution

Credentials and securing strings

Many administrators put passwords into the body of the script

Get-Credential and Read-Host, two ways to supply username and password

ConvertTo-SecureString – convert plain text or encrypted standard strings into a SecureString object

ConvertFrom-SecureString – convert secure strings into encrypted standard strings

Uses Windows Data Protection API ([DPAPI](#)) to encrypt/decrypt strings

Not foolproof, but it's pretty good

Works only for the same user on the same computer

To use stored credentials from any machine use Key/SecureKey, the Advanced Encryption Standard

Demo Secure Strings

Securing Remote sessions

Unable to take advantage due to outdated security and risk avoidance policies

Remoting enables only Adminis to connect, and they can only run commands they have permission to run

With Server 2012, Remoting is enabled by default and is mandatory for server management.

Background jobs

A command may take a long time to execute

Push the long-running command into the background

Background job creates new session that cannot be seen but execute command within it

Each job has one parent and one or more child jobs, which does the actual work

Using the Get-WMIObject and Invoke-Command to create background jobs

Stopping and Waiting for Jobs

- Stop-Job : Stops a background job
- Wait-Job : Suppresses command prompt until one or all of background jobs running in the session are complete

Demo Background Jobs

Day 3 Agenda

Default Values for Command Parameters

Error Handling

Managing Active Directory

Scripts

Snap-ins and Modules

Commands are shared by using Modules and Snap-ins

Modules and Snap-ins are packages containing PowerShell commands and other items

After running the module setup program or saving the module to disk, cmdlets and items in the module, can be used

Snap-in needs to be installed before cmdlets or providers in it can be used

Snap-in can only contain cmdlets and providers, a Module can also contain other common PowerShell items such as functions, variables, aliases and PowerShell drives

Snapins

Old method to extend the shell, from V1

Snapins always have to be installed and registered with the operating system

PSSnapins are saved in the registry

- Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\PowerShellSnapIns\

Disadvantages

- PSSnapins to be written in .NET and available as an Assembly
- Assemblies to be installed with a installutil.exe

Get a list of installed snapin names: *Get-PSSnapin -registered*

Load a snapin by name: *Add-PSSnapin name*

Show the commands in a snapin: *Get-Command -pssnapin name*

Show loaded snapins: *Get-PSSnapin*

Modules

Beginning with V3, modules are imported into the session automatically the first time a cmdlet in the module is used.

PSM1 – Windows PowerShell module file

Writing the first Module, Demo-Module

New-ModuleManifest

Demo Modules

Modules

Module is a set of related PowerShell functionalities, grouped together as a convenient unit

Allow the modularization of Windows PowerShell code

A module is made up of four basic components:

- Some sort of code file – usually either a PowerShell script
- Anything else that the above code file may need, such as additional assemblies, help files, or scripts
- Manifest file that describes above files, stores metadata such as author and versioning information
- Directory that contains all the above content, and is located where PowerShell can reasonably find it

Writing the first Module, Demo-Module

Script Module, Binary Module, Manifest Module and Dynamic Module

Demo Modules

Error Handling

'Terminating' and 'Non-Terminating' errors

Global variables - \$Error and -ErrorVariable

- **Get-Item C:\ -ErrorVariable err**
- **\$err.GetType().FullName**
- **\$err.count**
- **Get-Item c:\doesntexist.txt -ErrorVariable err -erroraction SilentlyContinue**
- **\$err[0]**
- **Error[0]**
- **Error[0] | GM**
 - *\$error[0].InvocationInfo* provides details about context which command was executed, if available
 - *\$error[0].Exception* contains original exception object as it was thrown to PowerShell
 - *\$error[0].Exception | gm*
 - *\$error[0].Exception.StackTrace*

\$LASTEXITCODE and **\$?**

Error Handling

-ErrorAction (-ea)

- SilentlyContinue (0)– error messages are suppressed and execution continues
- Stop – forces execution to stop, behaving like a terminating error
- Continue – the default option. Errors will display and execution will continue
- Inquire – prompt the user for input to see if we should proceed
- Ignore – (new in v3) – the error is ignored and not logged to the error stream

Preference variable - \$ErrorActionPreference

-ErrorAction will override the \$ErrorActionPreference

When -ErrorVariable parameter is used, the \$Error variable is still gets updated

- **Stop-Process -Name invalidprocess -ErrorVariable ProcessError**
- **Stop-Process -Name invalidprocess2 -ErrorVariable +ProcessError**

Error Handling

Try-Catch-Finally or Trap

- Handles terminating errors in scripts

Try - defines section of script you want to monitor for errors

Error is first saved to the **\$Error** automatic variable

Then **Catch** is searched to handle error

If not found then Parent scope is searched for Catch

After Catch is complete or if no Catch block or Trap statement is found, **Finally** is run

Try statement can include multiple Catch blocks for different kinds of errors

To handle Errors from non-PowerShell (Robocopy) processes use **\$LastExitCode**

Try – Catch - Finally

```
try
{
    $wc = new-object System.Net.WebClient
    $wc.DownloadFile("http://www.contoso.com/MyDoc.doc")
}
catch [System.Net.WebException],[System.IO.IOException]
{
    "Unable to download MyDoc.doc from http://www.contoso.com."
}
catch
{
    "An error occurred that could not be resolved."
}
finally
{
    "Completed"
```

Demo Error Handling

Managing Active Directory

Prerequisites

Connecting to Active Directory using the ADSI Provider

The Active Directory Name space

Creating Active Directory Structure using Organization Units

Creating, Modifying, and Deleting User Accounts



AD - Prerequisites

Install AD Module as part of Windows 2008R2 or Windows 2012

AD module requires at least Server 2008 R2 DC in environment

AD Web Services service needs to be running on at least one DC in environment

Windows PowerShell and the .NET Framework 3.5.1 or 4.5 must be installed

To manager AD from Windows 7 or 8, install Remote Server Administration Tools

Connecting to Active Directory

Import Active Directory Management Module, if not already installed

Tool used to connect is the Active Directory Services Interface (ADSI) Provider

- \$domain = [ADSI]"LDAP://dc=domain,dc=local"
- \$users = [ADSI]"LDAP://cn=Users,dc=domain,dc=local"
- \$users.Children

Use the LDAP ADSI Provider as the connector

Active Directory PSDrive

- cd AD:
- cd '.\DC=Domain,DC=local'
- cd '.\CN=Users
- Dir
- To return a list of groups from within the Users container
 - Get-ChildItem | Where-Object {\$_.ObjectClass -eq "Group"}

First Few Commands

Get-ADDomain provides you with a list of information about your domain

Get-ADUser <username> yields information about the user, if it exists

Get-ADUser -Filter {GivenName -eq "JohnDoe"}

Get-ADUser -Filter {Surname -eq "Doe"}

Get-ADPrincipalGroupMembership -Identity johndoe ????

New-ADUser -Name "Jane D" -GivenName Jane -Surname D -UserPrincipalName jane@test.local -SamAccountName Jane

Search-ADAccount -AccountDisabled -UserOnly | FT Name

Unlock-ADAccount -Identity (Read-Host "Username")

Get-ADDomain "Domain" -Server "Comp1" -Credential "Domain\Admin"

Get-ADUser naren -Server "Comp1" -Credential "Domain\Admin"

Get-ADUser -filter "department -eq 'sales'" | Disable-ADaccount

First Few Commands

Set-ADAccountPassword username -NewPassword (ConvertTo-SecureString -AsPlainText -String "P@ssw0rd1z3" -force)

Set-ADUser username -ChangePasswordAtLogon \$True

Get-ADUser -filter "enabled -eq 'false'" -property WhenChanged -SearchBase "DC=Users,DC=Domain,DC=Local" | where {\$_.WhenChanged -le (Get-Date).AddDays(-180)} | Remove-ADUser -whatif

Add-ADGroupMember "Bangalore Employees" -member (get-aduser -filter "city -eq 'Bangalore'")

Get-ADComputer -Filter * -Properties OperatingSystem | Select OperatingSystem -unique | Sort OperatingSystem

Active Directory Name Space

ADUC hides the actual LDAP names, methods, and properties utilized by AD

- LDAP Distinguished Name – CN=John Doe,OU=Sales,DC=DomainName,DC=com
- LDAP Relative Distinguished Name – John Doe
- Common Name – John Doe
 - version one – DC=com/DC=DomainName/OU=Sales/CN=John Doe
 - version two – DomainName.com/Sales/John Doe
- User Principal Name – John.Doe@DomainName.com
- Down level Name – DomainName\jdoe -or- jdoe

Get familiar to ADSIEdit

Demo – Connecting to AD Domain

Using New-Object

Using ADSI

Using various scripts

- Basic script
- Connecting as different user
- Connecting as different user, prompting User Credentials
- And few more examples

ADPowerShell_QuickReference.pdf and ADUsermanagement-powershell.pdf

Scripts

Create the script and save with a .PS1 file extension (e.g. myscript.ps1)

Run the script (c:\scripts\myscript.ps1), or (. \myscript.ps1)

If the path to the script contains a space, &"D:\my train\myscript.ps1"

Run a script by dot-sourcing it:

- . "D:\my train\My first Script.ps1"
- . .\Myscript.ps1"

Scripts - Parameters

```
Get-ChildItem -Path $env:windir -Filter *.dll -Recurse
or
```

```
Get-ChildItem `
-Path $env:windir `
-Filter *.dll `
-Recurse
```

```
$myargs = @{
  Path = "$env:windir"
  filter = '*.dll'
  Recurse = $true
}
```

```
Get-ChildItem @myargs
```

Script - Parameters

Start the script with

```
Param(
    [string]$computerName,
    [string]$filePath
)
```

```
.\Get-Something -computerName SERVER1 -filePath C:\Whatever
```

```
.\Get-Something -comp SERVER1 -file C:\Whatever
```

```
.\Get-Something SERVER1 C:\Whatever
```

```
.\Get-Something -filePath C:\Whatever -computerName SERVER1
```

Script - parameters

```
[CmdletBinding()]  
Param(  
    [Parameter(Mandatory=$True,Position=1)]  
    [string]$computerName,  
    [Parameter(Mandatory=$True)]  
    [string]$filePath  
)
```

Declaring as [string[]] accepts entire collection of values then enumerate using Foreach loop

Comma



Script - Parameters

Get-FreeDriveSpace script



WMI – Getting Started

Get-WmiObject is the key command


- Get-WmiObject -Class [classname] -NameSpace [namespace] -ComputerName [ComputerName] – Filter - Query

WMI classes are organized in the **Namespace**

- Get-WmiObject -List | ? {\$_.name -Match "Win32"}

Default namespace is **ROOT\cimv2**

Get-WmiObject -Class "Win32_NetworkAdapterConfiguration" | Get-Member -MemberType method



WMI Perspective – Hidden Treasure

Imagine WMI as a database, which keeps information about a computer's components such as the: BIOS, Services and Network Settings

Regard WMI as a method for collecting data about a computer's hardware and software

View WMI as a pipe, which magically connects to the core of any Microsoft operating system

Think of WMI as having its own PowerShell dialect, for example the WQL select clause

Treat WMI as a microscope, and use it to probe the operating system's objects and their properties

Not all classes are available on all versions of Windows



Demo – WMI

Introduction to WMI

Managing Remote Services



WMI Making changes to system - Example 1

```
$NICs = Get-WmiObject Win32_NetworkAdapterConfiguration | Where {$_.IPEnabled  
-eq "TRUE"}
```

```
foreach($NIC in $NICs) {$NIC.EnableDHCP(); $NIC.SetDNSServerSearchOrder()}
```

WMI Making changes to system – Example 2

```
$NICs=Get-WmiObject Win32_NetworkAdapterConfiguration|where{$_.IPEnabled -eq "TRUE"}
```

```
Foreach($NIC in $NICs) {
```

```
$NIC.EnableStatic("192.1.1.4", "255.255.255.0"); NIC.SetGateways("192.1.1.1")
```

```
$DNSServers = "198.1.2.2","198.1.2.6"; $NIC.SetDNSServerSearchOrder($DNSServers)
```

```
$NIC.SetDynamicDNSRegistration("TRUE"); $NIC.SetWINSServer("198.2.2.5", "198.1.2.6") }
```

- IP Addresses entered as String
- DNSSearchOrder is entered differently than WINSServer
- DNSRegistration is entered only as TRUE
- Get-WmiObject Win32_NetworkAdapterConfiguration | Get-Member -MemberType Methods | Format-List

WMI Making changes to system – Example 3

```
Get-WmiObject Win32_OperatingSystem | Get-Member -MemberType Method | Format-List
```

Local Reboot

```
$collItems = Get-WmiObject Win32_OperatingSystem
Foreach($Item in $collItems) { $Item.Reboot() }
```

Remote Reboot

```
$strComputer = Read-Host "Enter Computer Name"
$collItems = Get-WmiObject Win32_OperatingSystem -ComputerName "$strComputer"
Foreach($Item in $collItems) { $Item.Reboot() }
```

Use this script only to resolve issues, do not use it to flame your fellow coworkers