Microsoft

PowerShell Advanced Workshop

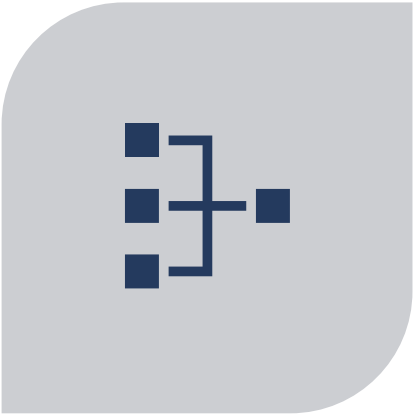
Module 1: PowerShell Foundation Skills Review

Object Models

What is an object?

STRUCTURED DATA

COMBINES SIMILAR INFORMATION AND CAPABILITIES INTO ONE ENTITY



A COLLECTION OF PARTS AND HOW TO USE THEM

How Would You Model a TV?

Properties (Information)
Is it on?
Current Channel
Current Volume
Screen Size
Brand
Input
Screen Type

Methods (Actions)
Toggle Power
Channel Up
Channel Down
Volume Up
Volume Down
Change Input
Set Channel(<int>)

To change the channel to a particular one we have to pass in data (the channel number).

Understanding Instances

Type [Microsoft.TV]	
Members	
Properties	Methods
DisplayType	VolumeUp()
Input	VolumeDown()
Size	ChannelUp()
ModelNumber	TogglePower()
…	…

$MyTv1	
Property	Value
DisplayType	LCD
Input	VGA
Size	42
ModelNumber	PTV-42732
…	…

$MyTv2	
Property	Value
DisplayType	LED
Input	HDMI1
Size	80
ModelNumber	LEDTV-80432
…	…

Object-Based Shell

Everything is represented as an OBJECT

An OBJECT is an INSTANCE of a TYPE

OBJECTS have data fields (PROPERTIES) and procedures (METHODS)

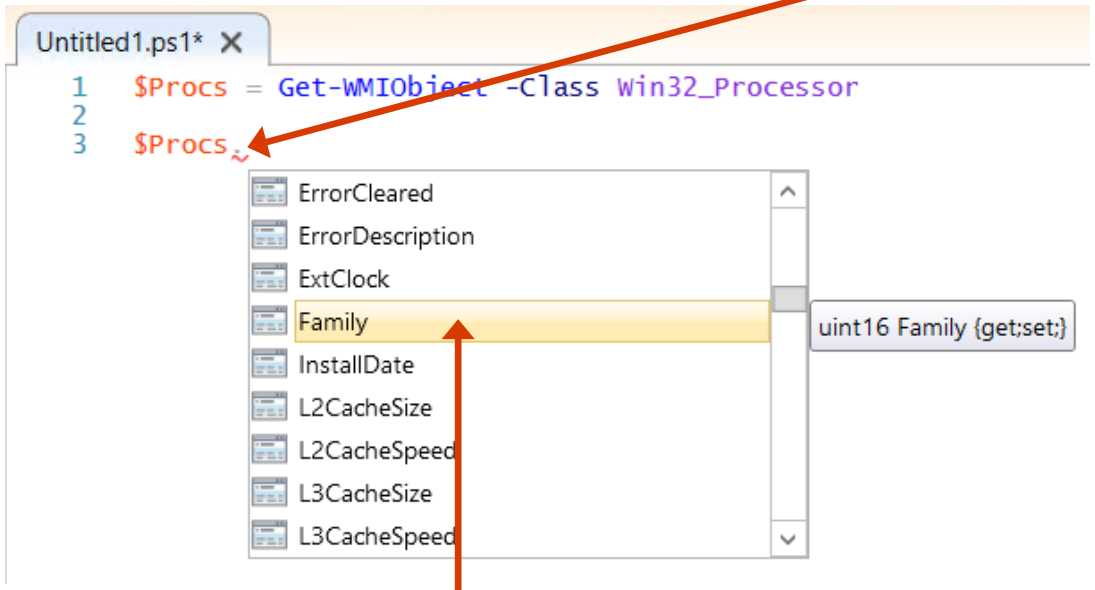
A TYPE represents a construct that defines a template of MEMBERS

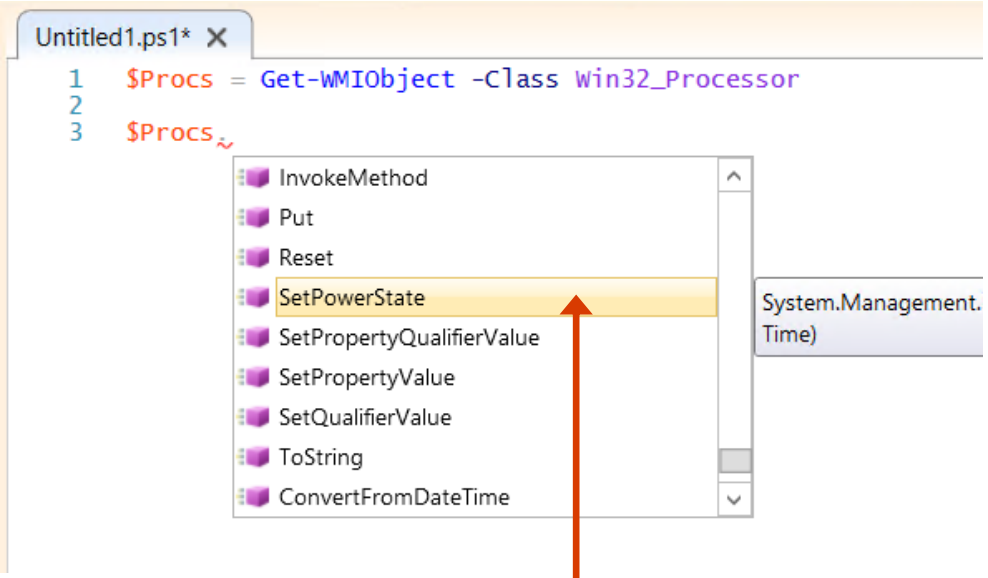
PROPERTIES and METHODS are collectively known as MEMBERS

Accessing Members – ISE

ISE IntelliSense

Type "." to access members





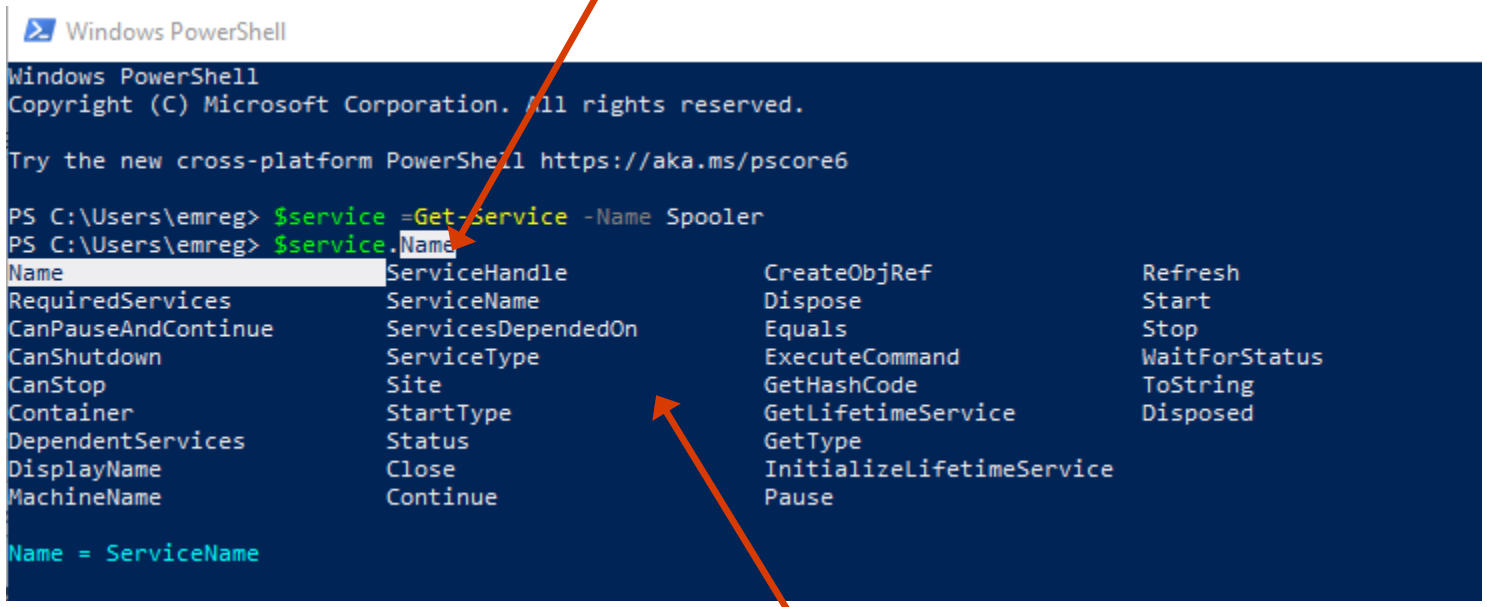
Properties

Methods

Accessing Members – Console

Console IntelliSense

Type "." then CTRL + Space



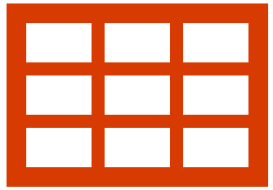
Properties & Methods

Identify PROPERTIES and METHODS for an object

Why Should discover / identify methods and properties

Take Action

Methods are ready to use functions. You can take action immediately.



Parse Less

Properties are structured data, you don’t need to parse the results in most cases.

Get-Member Overview

Discover Properties and methods of an Object

- Displays PROPERTIES and Methods
- Shows the Type of the Object
- PROPERTIES are columns of Information
- METHODS are actions that can be taken on the object

Get-Member cmdlet

Shows the **type** name, **properties** and **methods**

The object is passed to **–InputObject** parameter

```
PS>  Get-Member -InputObject "Some String"

 TypeName: System.String

Name          MemberType            Definition
----          ----------            ----------
Trim          Method                string Trim(Params char[] trimChars)....
Length        Property              int Length {get;}
```

Get-Member Property Definition

```
PS C:\> $item = Get-Item C:\Windows\System32\drivers\etc\hosts
PS C:\> Get-Member -inputobject $item -Name LastWriteTime

 TypeName: System.IO.FileInfo

Name            MemberType          Definition
----            ----------          ----------
LastWriteTime   Property            datetime LastWriteTime {get;set;}
```

Data type: **[DateTime]**

Can be **get** (received) or **set** (changed)

```
PS C:\> $file = Get-Item C:\Windows\System32\drivers\etc\hosts
PS C:\> $file.LastWriteTime = (Get-Date)
PS C:\> Get-Item C:\Windows\System32\drivers\etc\hosts
Directory: C:\Windows\System32\Drivers\etc

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12/23/2020   4:23 PM            894 hosts
```

Get-Member Method Definition

```
PS C:\> $file = Get-Item C:\Windows\notepad.exe
PS C:\> Get-Member -InputObject $file -Name CopyTo

 TypeName: System.IO.FileInfo

Name     MemberTyp    Definition
----     ----------   ----------
CopyTo   Method       System.IO.FileInfo CopyTo(string destFileName),
                      System.IO.FileInfo CopyTo(string destFileName, bool ov..
```

Two parameter sets

This Method **returns** a **System.IO.FileInfo**

```
PS C:\> $file = Get-Item C:\Windows\notepad.exe
PS C:\> $file.CopyTo("C:\Temp\notepad.exe", $True)

 Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        7/16/2016   7:43 AM         243200 notepad.exe
```

The newly copied file is **System.IO.FileInfo**

Variables

Variables Overview

What are variables?

User-Defined Variables

Working with Strings

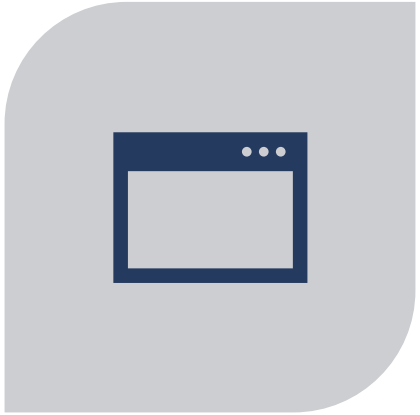
What Are Variables?

- Unit of memory
- Defined and accessed using a dollar sign prefix ($)
- Holds an object which can also be a collection of objects
- Variable names can include spaces and special characters
- Not case-sensitive

- Kinds of variables:
  - Automatic (built-in)
  - User-defined

Automatic Variables

BUILT-IN



CREATED AND MAINTANED BY POWERSHELL

STORE POWERSHELL STATE

Automatic Variables Examples

Get-Help about_Automatic_Variables

Type	Example
List of all errors	`PS C:\> $Error`
Execution status of last operation	`PS C:\> $?`
User's home directory	`PS C:\> $HOME`
Current host application for PowerShell	`PS C:\> $Host`
NULL or empty value	`PS C:\> $null`
Full path of installation directory for PowerShell	`PS C:\> $PSHOME`
Represents TRUE in commands	`PS C:\> $true`
Represent FALSE in commands	`PS C:\> $false`

User-Defined Variables

EXISTS ONLY IN CURRENT SESION

CREATED AND MAINTANED BY USER

LOST WHEN SESSION IS CLOSED

Creating User Defined Variable

Assignment Operator '**=**'

-OutVariable common parameter

Variable **Cmdlets**

```
PS C:\> $svcs = Get-Service

PS C:\> Get-Service -OutVariable svcs

PS C:\> New-Variable -Name svcs -Value (Get-Service)

PS C:\> $svcs

Status   Name         DisplayName
------   ----         -----------
Stopped  AeLookupSvc  Application Experience
Stopped  ALG          Application Layer Gateway Service
Running  AppIDSvc     Application Identity
Running  Appinfo      Application Information
...
```

Subexpression

Expressions within Expressions instead of user-defined variables

- Can be used as in line expressions
- Avoids using unnecessary variables
- Can be nested
- The expression within, returns object or objects

```
# two lines of code
$Service = Get-Service -Name Spooler
Get-Member -InputObject $Service

# less line of code
Get-Member -InputObject (Get-Service -Name Spooler)

# Can access properties as well
(Get-Service -Name Spooler).
```

CanPauseAndContinue
CanShutdown
CanStop
Container
DependentServices
DisplayName
MachineName
Name
PSStandardMembers

Variable Cmdlets

Name	Example
New-Variable	`PS C:\> New-Variable zipcode -Value 98033`
Clear-Variable	`PS C:\> Clear-Variable -Name Processes`
Remove-Variable	`PS C:\> Remove-Variable -Name Smp`
Set-Variable	`PS C:\> Set-Variable -Name desc -Value "Description"`
Get-Variable	`PS C:\> Get-Variable -Name m*`

Constant Variables

- Variables can only be made constant at creation (cannot use “=”)
- Cannot be deleted
- Cannot be changed

```
PS C:\> New-Variable -Name pi -Value 3.14159 -Option Constant
```

ReadOnly Variables

- Cannot mark a variable ReadOnly with “=“
- Cannot be easily deleted (must use Remove-Variable with -Force)
- Cannot be changed with “=“ (must use Set-Variable with -Force

```
PS C:\> New-Variable -Name max -Value 256 -Option ReadOnly
```

Objects and Variables

Summary

Always keep in mind, Everything is OBJECT in PowerShell

Each Object Has a TYPE

Variables reference OBJECTS

Pipeline Introduction

What is a Pipeline?

Series of commands connected by pipeline character

Vertical bar character |

Sends output of command as input to another (left to right)

Passes Objects, not text

Filtering, Formatting, and Outputting available

Cmdlets designed to chain together into ‘pipelines’

Get cmdlets

- Typically placed first in the pipeline
- Provides the input to be processed

Returns all services

```
PS C:\>  Get-Service | Export-Csv C:\temp\services.csv
```

Takes an action on the services of creating text file

File input

Text files provide input to be processed by pipeline

Reads file

```
PS C:\> Import-Csv .\services.csv | Select-Object DisplayName
```

Selects each object on each line in file

```
DisplayName
-----------
Agent Activation Runtime_28896f
AllJoyn Router Service
Application Layer Gateway Service
Application Identity
...
```

Pipeline Object Manipulation

Object cmdlets

Sort-Object

- Sorts objects by property values

Select-Object

- Selects object properties

Group-Object

- Groups objects that contain the same value for specified properties

Measure-Object

- Calculates numeric properties of objects
- Ex. characters, words, lines in string objects

Compare-Object

- Compares two sets of objects

Sort-Object and Select-Object

Get all processes, **Sort** by handle counts, then **Select** bottom 2

```
PS C:\> Get-Process | Sort-Object -Property Handles | Select-Object -last 2

Handles  NPM(K) PM(K)  WS(K)  VM(M)  CPU(s) Id  ProcessName
-------  ------ -----  -----  -----  ------ --  -----------
   1283      55 21020  30340   1237  477.78 304  svchost
   1926      44 285244 230112  1165  716.45 4124 livecomm
```

Storing pipeline output in variable

- Pipeline output can be stored in a user-defined variable using the "**=**" assignment operator

Storing cmdlet output in a variable

```
PS C:\> $Events = Get-EventLog -LogName Security | Group-Object EntryType
```

Accessing output using variable **name** and **$** prefix

```
PS C:\> $Events

 Count Name          Group
 ----- ----          -----
135950 SuccessAudit {System.Diagnostics.EventLogEntry...
    40 FailureAudit {System.Diagnostics.EventLogEntry...
```

Out-GridView

```
PS C:\> Get-Process | Out-GridView
```

Get-Process | Out-GridView

Filter

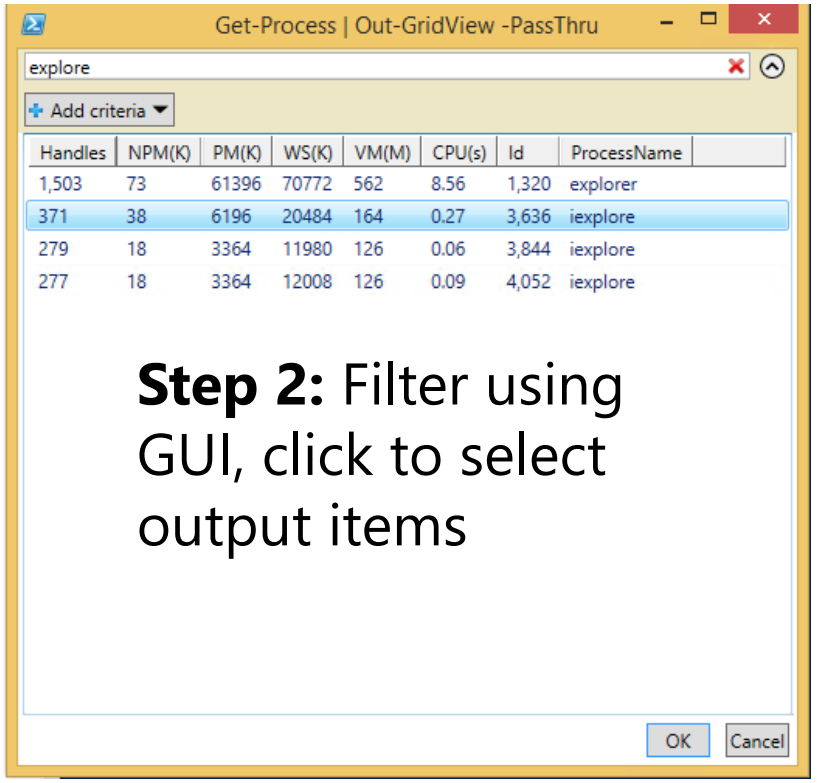
Add criteria

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
589	33	29724	32988	2.94	3,004	1	ApplicationFrameHost
736	61	82816	59300	406.48	11,536	1	atmgr
231	14	8640	15008	35.75	15,716	0	audiodg
573	28	20824	31808	0.48	3,956	1	Calculator
1,599	49	39688	47272		14,892	0	CcmExec
419	20	4652	11060	35.53	2,492	1	ciscowebexstart
143	7	1632	5876	0.27	14,196	1	CompPkgSrv
836	28	1912	3012		848	0	csrss
848	38	4280	4968		972	1	csrss
701	20	58776	29688	101.92	11,716	1	ctfmon
410	19	6868	9500		5,032	0	dasHost
146	9	1980	8444	0.80	1,756	1	dllhost
208	17	3444	6812		6,664	0	dllhost
372	29	7964	12388	2.19	12,612	1	dllhost
260	14	4032	12912		17,952	0	dllhost
53	6	1344	3164	11.03	9,432	1	dptf_helper

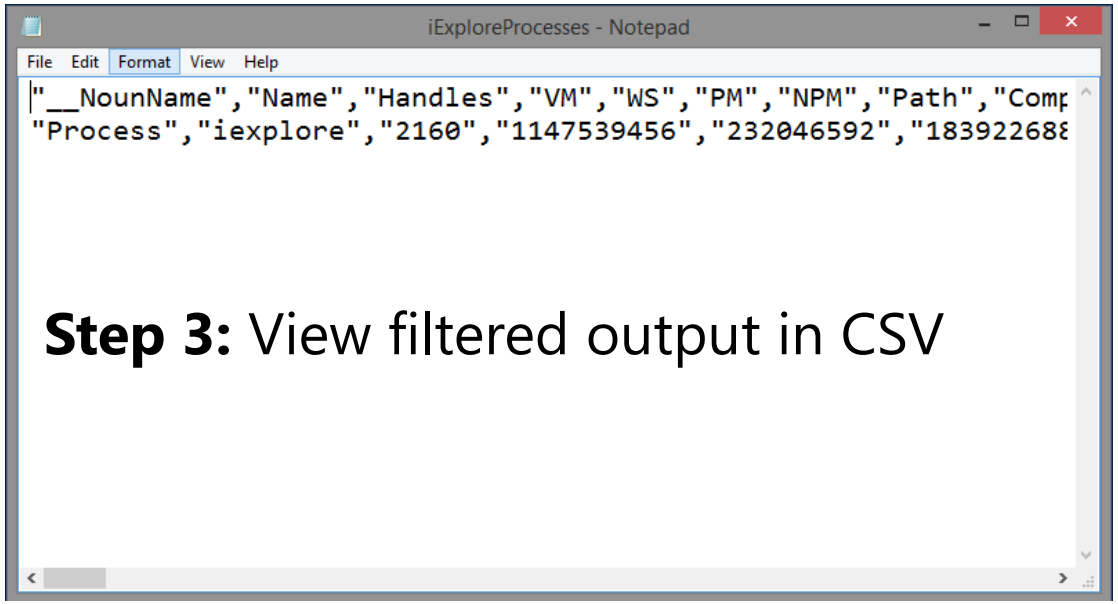
Out-GridView with PassThru

Step 1: Create variable with content and pipe to desired result

```
PS> $Procs = Get-Process
PS> $Procs | Out-GridView -PassThru | Export-Csv c:\temp\File.csv –NoTypeInformation
```



Step 2: Filter using GUI, click to select output items



Step 3: View filtered output in CSV

ConvertTo/From cmdlets

ConvertTo/From cmdlets

Helpful when converting native data formats into PowerShell objects

ConvertTo-CSV
ConvertFrom-CSV

ConvertTo-Json
ConvertFrom-Json

ConvertTo-Html

ConvertTo-Json

```
PS C:\> Get-Service | ConvertTo-Json | Out-File c:\temp\services.json
PS C:\> notepad.exe C:\temp\services.json
PS C:\> code . C:\temp\services.json
```


```
[
  {
    "CanPauseAndContinue": false,
    "CanShutdown": false,
    "CanStop": false,
    "DisplayName": "Agent Activation Runtime_28896f",
    "DependentServices": [
                    ],
    "MachineName": ".",
    "ServiceName": "AarSvc_28896f",
    "ServicesDependedOn": [
                    ],
    "ServiceHandle": null,
    "Status": 1,
    "ServiceType": 224,
    "StartType": 3,
    "Site": null,
    "Container": null,
    "Name": "AarSvc_28896f",
    "RequiredServices": [
                    ]
  },
```



```
[
    {
        "CanPauseAndContinue": false,
        "CanShutdown": false,
        "CanStop": false,
        "DisplayName": "Agent Activation Runtime_28896f",
        "DependentServices": [
                                ],
        "MachineName": ".",
        "ServiceName": "AarSvc_28896f",
        "ServicesDependedOn": [
                                ],
        "ServiceHandle": null,
        "Status": 1,
        "ServiceType": 224,
        "StartType": 3,
        "Site": null,
        "Container": null,
        "Name": "AarSvc_28896f",
        "RequiredServices": [
                                ]
    },
```

Pipeline Advanced

Learnings covered in this Unit

- Pipeline variables
- Filtering on the pipeline
- Looping elements in the pipeline
- Pipeline input

Pipeline Variable

Pipeline Variable Overview

- Represents the **current** object on the pipeline
- Used perform an action on **every** object
- Used with cmdlets like **Foreach-Object** and **Where-Object**
- **$_** and **$PSItem**
- Use **-PipelineVariable** parameter to name your own variable on pipeline
- Scoped only to **current** pipeline

Object Cmdlets

ForEach-Object

- Performs an **operation** against **each object** on the pipeline
- Aliases: **%** and **ForEach**

Where-Object

- **Filters** objects in pipeline using a **script block** to check **conditions**
- Aliases: **?** and **Where**

Foreach-Object

Foreach-Object Basics

Performs an **action** to **every** object on the pipeline using a **script block**

Aliases: **%** and **Foreach**

Script block can perform **any amount** of code and be **saved** into a **variable**

$_ allows accessing **properties** or **methods**

Automatic Member Enumeration

Retrieve **single property** from collection **without** using ForEach-Object

```
PS> (Get-Process).ID
4300
8844
8812
```

Multiple levels deep

```
PS> (Get-EventLog -Log System).TimeWritten.DayOfWeek | Group-Object

Count Name      Group
----- ----      -----
 4174 Tuesday   {Tuesday, Tuesday, Tuesday...}
 4349 Monday    {Monday, Monday, Monday...}
```

Foreach-Object Example: Active Directory

The .. operator will return each integer between the two values

Each integer is passed through the pipeline to ForEach-Object

ForEach-Object will use the **$_** variable to represent each integer in the following commands

```
PS> 1..100 | ForEach-Object {
    New-ADUser  -Name User$_ `
      -Organization "contoso.com/Accounts" `
      -UserPrincipalName "User$_@contoso.com" `
      -emailaddress "User$_@contoso.com" `
      -ChangePasswordAtLogon $true
}
```

Where-Object

Where-Object Filtering

Script block needs to return **True** or **False**

$_ allows accessing **properties** or **methods**

Comparison and **Logical Operators** are generally used

Any value except **$False**, **$Null**, and **0** considered True

Where-Object Basics

Filters objects on pipeline using a **script block** to check **conditions**

Aliases: **?** and **Where**

```
PS> Get-Service | Where {$_.CanPauseAndContinue}

Status   Name                DisplayName
------   ----                -----------
Running  LanmanWorkstation   Workstation
Running  QualysAgent         Qualys Cloud Agent
Running  TechSmith Uploa...  TechSmith Uploader
Running  Winmgmt             Windows Management
```

Boolean property is already **True** or **False**

4 services returned instead of all 300

Comparison Operators

	Case Insensitive	Case Sensitive
Equal	-eq	-ceq
Not Equal	-ne	-cne
Greater Than	-gt	-cgt
Greater Than or Equal To	-ge	-cge
Less Than	-lt	-clt
Less Than or Equal To	-le	-cle

No Wildcards

	Case Insensitive	Case Sensitive
Equal With Wildcard	-like	-clike
Not Equal With Wildcard	-notlike	-cnotlike

Wildcards

More comparison operators will appear in other sections

Basic Comparison Examples

```
PS> "This" -eq "That"
False

PS> "This" -eq "This"
True

PS> "This" -eq "Th*"
False

#Wildcard must be on right
PS> "This" -like "Th*"
True

PS> "This" -like "That"
False

PS> "This" -notlike "That"
True
```

```
PS> 5 -gt 3
True

PS> 5 -gt 5
False

PS> 5 -ge 5
True

#Case Insensitive
PS> "This" -eq "this"
True

#Case Sensitive
PS> "This" -ceq "this"
False
```

Where-Object Using Comparisons

- Use pipeline variable: **$_** or **$PSItem**
- Compare **properties** or **methods** output to other **values**

```
PS> Get-Service | Where-Object {$_.StartType -eq "Disabled"}

 Status   Name               DisplayName
------   ----               -----------
Stopped  AppVClient         Microsoft App-V Client
Stopped  NetTcpPortSharing  Net.Tcp Port Sharing Service
Stopped  RemoteAccess       Routing and Remote Access
Stopped  RemoteRegistry     Remote Registry
Stopped  shpamsvc           Shared PC Account Manager
Stopped  ssh-agent          OpenSSH Authentication Agent
Stopped  tzautoupdate       Auto Time Zone Updater
Stopped  UevAgentService    User Experience Virtualization Service
```

Logical Operators – Basic – 1

Join **multiple comparisons** together into **compound conditions**

Operator	Description
-and	**TRUE** only when **both statements** are TRUE
-or	**TRUE** when **either** or both statements are TRUE
-xor	**TRUE** only when **one** of the statements is **TRUE** and the **other** is **FALSE**
-not	**Prepended - Toggles** the statement TRUE to FALSE or vice versa
!	Same as -not

Where-Object Simple Syntax

Shortcut for **simple** comparisons

PowerShell **v3.0**+

Compound conditions need **full syntax**

Full syntax

```
PS> Get-Service | Where-Object {$_.Status -eq "Running"}
```

Simple syntax

```
PS> Get-Service | Where Status -eq Running
```

Full syntax needed for compound conditions

```
PS> Get-Service | Where-Object {$_.Status -eq "Running" -and $_.CanStop}
```

Filtering with Parameters vs. Where-Object

If a cmdlet has a **parameter** to **filter** upon, it is usually **optimized**

Where-Object is a great **backup**, but always check the cmdlet's parameters first

Observable with **large data sets**, but negligible with small data sets

Filter output with Where-Object (~11 milliseconds)

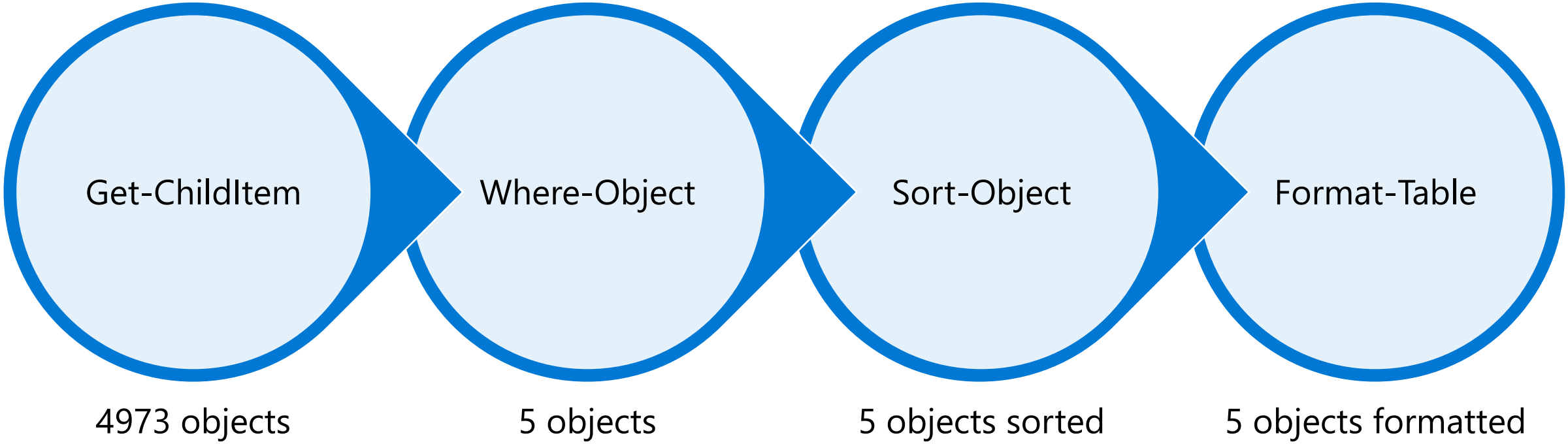
```
PS> Get-Process | Where-Object {$_.Name -eq "explorer"}
```

Filter output with parameters (~4 milliseconds)

```
PS> Get-Process -Name explorer
```

Piping

```
PS>  Get-ChildItem -Path C:\Windows\System32 |
       Where-Object Length -gt 50mb |
         Sort-Object Length |
             Format-Table Name,Length
```



Pipeline Processing with Foreach and Functions

Foreach-Object -Process Parameter

ForEach-Object is often used with a positional parameter in simple scenarios

Other parameters exist for specialized processing

```
PS C:\> Get-EventLog -LogName Application -Newest 5 |
ForEach-Object {$_.Message | Out-File -Filepath Events.txt -Append}
```

Position 1 is -Process Parameter

```
PS C:\> Get-EventLog -LogName Application -Newest 5 |
ForEach-Object -Process {$_.Message | Out-File Events.txt -Append}
```

-Process parameter can be named

Parameters – Begin

- ForEach-Object cmdlet supports Begin, Process, and End Parameters

- Begin block → run once before any items are processed
- Process block → run for each object on pipeline
- End block → run once after all items have been processed

```
PS C:\> Get-EventLog -LogName Application -Newest 5 |
ForEach-Object
-Begin {Remove-Item .\Events.txt; Write-Host "Start" -ForegroundColor Yellow}
-Process {$_.Message | Out-File -Filepath Events.txt -Append}
-End {Write-Host "End" -ForegroundColor Green; notepad.exe Events.txt}
```

Parameters – Process

- ForEach-Object cmdlet supports Begin, Process and End Parameters

- Begin block → run once before any items are processed
- Process block → run for each object on pipeline
- End block → run once after all items have been processed

```
PS C:\> Get-EventLog -LogName Application -Newest 5 |
ForEach-Object
-Begin {Remove-Item .\Events.txt; Write-Host "Start" -ForegroundColor Yellow}
-Process {$_.Message | Out-File -Filepath Events.txt -Append}
-End {Write-Host "End" -ForegroundColor Green; notepad.exe Events.txt}
```

Parameters – End

- ForEach-Object cmdlet supports Begin, Process and End Parameters

- Begin block → run once before any items are processed
- Process block → run for each object on pipeline
- End block → run once after all items have been processed

```
PS C:\> Get-EventLog -LogName Application -Newest 5 |
ForEach-Object
-Begin {Remove-Item .\Events.txt; Write-Host "Start" -ForegroundColor Yellow}
-Process {$_.Message | Out-File -Filepath Events.txt -Append}
-End {Write-Host "End" -ForegroundColor Green; notepad.exe Events.txt}
```

Named Blocks in Functions/ScriptBlocks

Optional named blocks in a function

- Allows for **processing** collections from the pipeline
- Can be defined in **any** order

Begin Block

- Statements executed **once**, **before** first pipeline object

Process Block

- Statements **executed** for **each** pipeline **object** delivered, leveraging **$_**
- If called outside a pipeline context, block is executed exactly once
- Becomes more common and **useful** with **Advanced Functions**

End block

- Statements executed **once**, **after** last pipeline object

Named Blocks in Function

```
function My-Function
{
    Begin
    {
        Remove-Item .\Events.txt
        Write-Host "Start" -ForegroundColor Red
    }
    Process
    {
        $_.Message | Out-File -Filepath Events.txt -Append
    }
    End
    {
        Write-Host "End" -ForegroundColor Green
        notepad.exe Events.txt
    }
}
```

```
PS> Get-EventLog -LogName Application -Newest 5 | My-Function
```

Pipeline Input

Methods Of Accepting Parameter Pipeline Input

By Value

- Attempted first
- Incoming **object** and **parameter** are of **same** data TYPE
- Incoming **object** can be **converted** to same data **TYPE** as the parameter

By Property Name

- Attempted if object **does not** come in by **value**
- Incoming object has a **property name** that matches the **parameter name** and is the **same** data TYPE

Cmdlet parameters may **accept** pipelined **objects** by value, by property name or **both**.

Does a Parameter Accept Pipeline Input?

```
PS> Get-Help Restart-Computer -Parameter ComputerName

 -ComputerName <String[]>

    Specifies one or more remote computers. The default is ...

    Required?                    false
    Position?                    1
    Default value                Local computer
    Accept pipeline input?       True (ByValue, ByPropertyName)
    Accept wildcard characters?  false
```

Pipeline Input ByValue

```
PS>  Get-Help Get-Timezone -Parameter name

-Name <String[]>
    Specifies, as a string array, the name or names of the time zones
that this cmdlet gets.

    Required?                    false
    Position?                    0
    Default value                None
    Accept pipeline input?       True (ByValue)
    Accept wildcard characters?  false
```

Strings

```
PS> "Eastern Standard Time", "Mountain Standard Time" | Get-TimeZone

PS> "Eastern Standard Time", "Mountain Standard Time" |
        ForEach-Object {Get-TimeZone -name $_}
```

Same Results

Pipeline Input ByPropertyName

```
PS> Get-Help New-Alias -Parameter Name
  -Name <String>
    Required?                    true
    Accept pipeline input?       True (ByPropertyName)

PS> Get-Help New-Alias -Parameter Value
  -Value <String>
    Required?                    true
    Accept pipeline input?       True (ByPropertyName)
```

Aliases.csv - Notepad

File Edit Format View Help

```
Name,Value
P,Get-Process
S,Get-Service
ping,Test-Connection
```

```
PS> import-csv C:\temp\aliases.csv | GM

 TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Name        NoteProperty string Name=P
Value       NoteProperty string Value=Get-Process
```

Demonstration Pipeline Input

Pipeline Input

Functions

Function Overview

Reusable block of PowerShell code

Reduces size of code and increases reliability

Can accept **parameter** values and return output

Advanced Functions behave like **Cmdlets**, including help content

What Does a Function Look Like?

1. Function Keyword
2. Function Name
3. Curly Brace Pair
4. PowerShell Commands
5. Call the Function
6. Function output

```
1 2
Function Write-Statement
3 4
{
  Write-Host “Hello world!” -ForegroundColor Green
3
}

5 PS> Write-Statement

6 Hello World!
```

Creating a Function

Multiple commands can be contained within a function

Allows for large code blocks to be reused – no need to copy and paste

Maintains consistency between repeated uses of code when edited

```
Function Write-ServiceStatus
{
  $SVC = Get-Service -Name WinRM
  $Name = $SVC.DisplayName
  $Status = $SVC.Status
  Write-Host "The Service $Name is currently $Status" -ForegroundColor Green
}

PS> Write-ServiceStatus
The Service Windows Remote Management (WS-Management) is currently Stopped
```

Parameters

Parameters in a function

- Must be the **first line** of code in the function
- Defined using **param ( )** statement
- When used, the parameter name is preceded by a **hyphen**
- By default, accepts **any** data type, is **positional**, and is **optional**
- **Optional** advanced attributes can override default parameter behavior

Add Parameters to a Function

1. Function Keyword
2. Function Name
3. Curly Brace Pair
4. PowerShell Commands
5. Call the Function
6. Function output

```
Function Write-Statement
{
  Write-Host "Hello World!" -ForegroundColor Green
}

PS> Write-Statement

Hello World!
```

Add Parameters to a Function

1. Function Keyword
2. Function Name
3. Curly Brace Pair
4. Parameter statement
5. PowerShell Commands
6. Call the Function
7. Function output

```
Function Write-Statement
{
  Param($Statement)
  Write-Host $Statement -ForegroundColor Green
}

PS> Write-Statement -Statement "Hello World!"

Hello World!
```

Default Parameter Values

Can assign a value like any other variable

Function parameters will override the default value

```
Function Write-Statement
{
  Param($Statement = "Good morning!")
  Write-Host $Statement -ForegroundColor Green
}

PS> Write-Statement

Good morning!
```

Adding Multiple Parameters

Functions can accept multiple parameters separated by commas

Supports line breaks between parameters

```
Function Write-ServiceStatus
{
  Param ($Service,
         $Color = "Green")
  $SVC = Get-Service -Name $Service
  $Name = $SVC.DisplayName
  $Status = $SVC.Status
  Write-Host "The Service $Name is currently $Status" -ForegroundColor $Color
}

PS> Write-ServiceStatus  -Service WinRM -Color Yellow
The Service Windows Remote Management (WS-Management) is currently Stopped
```

Script Blocks

What is a Script Block?

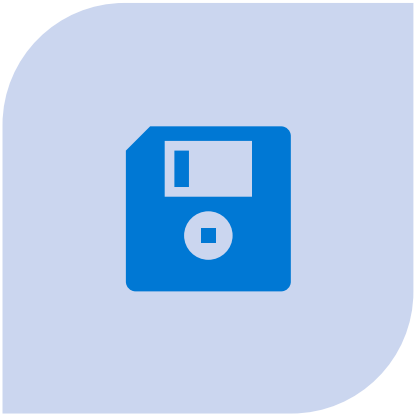
A collection of statements listed in curly brackets “{ }”

Used by Cmdlets, Functions, Automation, and other advanced features

Can accept parameter values and return output

* Script blocks will continue to be used throughout this course

When to a use script blocks



Save code for reuse with functions and automation

Remotely sending commands to another machine

Complex queries and filters on the pipeline

Using Script Blocks

Measure-Command uses the Expression parameter

- Expression parameter accepts the script block

Measures the time it takes commands to run

- Also executes the commands

Many other cmdlets have similar parameters that use Script Blocks

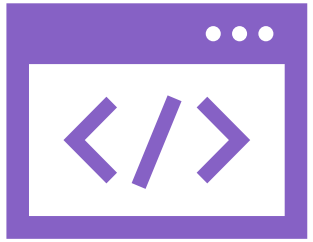
- Identify other commands with: *Get-Command –ParameterType ScriptBlock*

```
PS> Measure-Command -Expression {Get-Process}

Days              : 0
Minutes           : 0
Seconds           : 2
Milliseconds      : 933
```

Script Blocks

Script Blocks can be saved in variables

```
PS> $ScriptBlock = {Get-Service -Name WinRM}
PS> $ScriptBlock

Get-Service -Name WinRM
```



Script block code is contained but not executed

Invoking Script Blocks

Using a Cmdlet: **Invoke-Command –ScriptBlock $ScriptBlock**

Using a Method: **$ScriptBlock.Invoke()**

```
PS> $ScriptBlock = { Get-Service -Name WinRM }
PS> Invoke-Command -ScriptBlock $ScriptBlock

Status   Name   DisplayName
------   ----   -----------
Stopped  WinRM  Windows Remote Management (WS-Management)

PS> $ScriptBlock.Invoke()

Status   Name   DisplayName
------   ----   -----------
Stopped  WinRM  Windows Remote Management (WS-Management)
```

What are Scripts?

Text file (.ps1) containing one or more PowerShell commands

Simple ‘code packaging’ for distribution purposes and later use

Supports all features a function does:

- Accepts parameters
- Returns values
- Leverages help syntax

Can also be digitally signed for security

Microsoft

Scripts

Simple Script Example

Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Sample.ps1

```
1 Write-Host ‘Start of script’ -BackgroundColor Green
2 Write-Host ‘Display the % CPU Time utilization by the ISE’ -BackgroundColor Green
3 Get-Counter ‘\Process(powershell_ise)\% Processor Time’
4
5
6
7
8
9
10
11
12
```

```
PS C:\scripts> C:\scripts\Sample.ps1
Start of script
Display the % CPU Time utilization by the ISE

Timestamp                 CounterSamples
---------                 --------------
6/25/2018 11:22:39 AM     \\WIN10\process(powershell_ise)\% processor time :
                          14.0038797593275
```

Launching a script

Running Powershell Scripts

From Command Line:

Full path and file name

```
PS C:\> c:\scripts\script.ps1
```

Relative path

```
PS C:\Scripts> .\script.ps1
```

Spaces in path (use tab completion)

```
PS C:\> & "c:\scripts\my script.ps1"
```

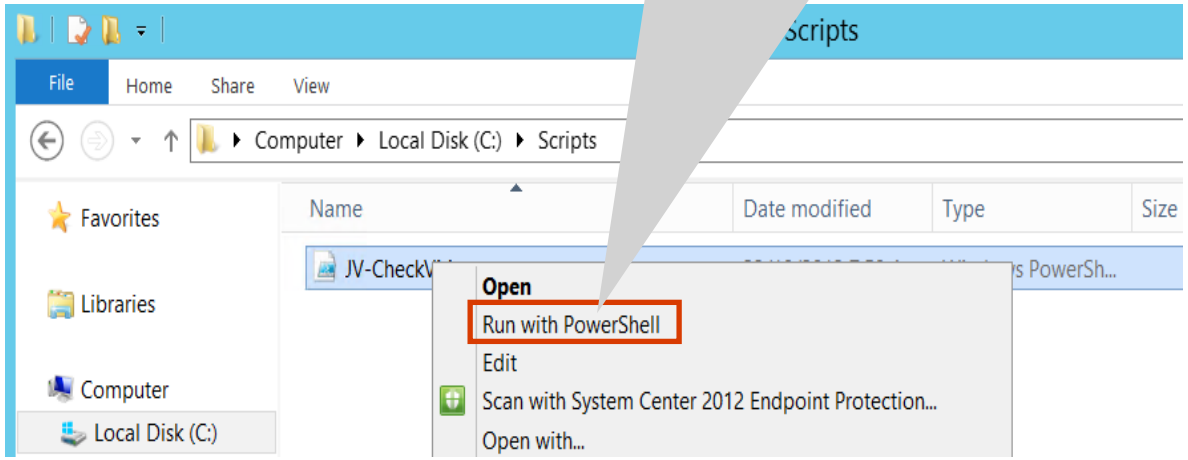
Script is in environment path

```
PS C:\> Script.ps1
```

From GUI:

Script files cannot be run by double-clicking

1. Right-click script
2. Select “Run with Powershell”



Script Param Statement

- Must be first statement in script, except for comments
- Parameter values are available to commands in scripts

Param Statement

ScriptParamExample.ps1

```
1  param ($ComputerName)
2
3  $result = Test-Connection -ComputerName $ComputerName -Quiet -Count 1
4
5  Write-Host $result -ForegroundColor Green
```

Parameter

Parameter Value

```
PS C:\scripts> .\ScriptParamExample.ps1 -ComputerName localhost
True

PS C:\scripts> .\ScriptParamExample.ps1 -ComputerName DoesNotExist
False
```

Execution Policies

Execution Policy

Determines conditions under which PowerShell will run scripts

Can be set for:

- Local computer
- Current user
- Specific Powershell session
- Group Policy computers and users

Not a full security system:

- Does **NOT** restrict user actions nor typing individual PS commands
- Helps users set basic rules for and prevents unintentional violations of the rules

Execution Policy Levels

Restricted - Default in all Client OS versions

- Scripts cannot be run
- PowerShell interactive-mode only

AllSigned

- Runs a script only if digitally signed with trusted certificate on local machine

RemoteSigned - Default in all Server OS versions (*Recommended Minimum)*

- Runs all local scripts
- Downloaded scripts must be signed by trusted source

Bypass

- Nothing locked, no warnings or prompts
- Used when script is built into larger application that has its own security model

Unrestricted

- All scripts can be run

Execution Policy Scope

AD Group Policy – Computer

- Affects all users on targeted computer

AD Group Policy – User

- Affects users targeted only

Process

- Command-line Parameter (*c:\> powershell.exe –executionpolicy remotesigned)*
- Affects current PowerShell Host session only

Registry – User

- Affects current user only
- Stored in HKCU registry subkey (Admin access **not** needed)

Registry – Computer

- Affects all users on computer
- Stored in HKLM registry subkey (Admin access **needed** to change)

Highest Priority Wins

Setting / Determining Execution Policy

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Unrestricted
```

Apply setting to **current** user only

```
PS C:\> Get-ExecutionPolicy -List

Scope            ExecutionPolicy
-----            ---------------
MachinePolicy    Undefined
UserPolicy       Undefined
Process          Undefined
CurrentUser      Unrestricted
LocalMachine     RemoteSigned

PS C:\> Get-ExecutionPolicy
Unrestricted
```

Topmost takes precedence

Effective Policy

Script Signing

Validates the integrity of the script

Enforced with **Execution Policies**

Certificate used should be of type **Code signing**

Signing a Script

Step 1: Create a certificate variable (2 ways)

Retrieve a code-signing certificate from the certificate provider

```
PS C:\> $cert = Get-ChildItem -Path Cert:\CurrentUser\My -CodeSigningCert
```

-- OR --

Find a code signing certificate

```
PS C:\> $cert = Get-PfxCertificate -Path C:\Test\MySign.pfx
```

Trusted by computer where script will run

Step 2: Sign the script

```
PS C:\> Set-AuthenticodeSignature .\ISECPUTime.ps1 $cert

   Directory: C:\Scripts

SignerCertificate                Status     Path
-----------------                ------     ----
A4..                             Valid      ISECPUTime.ps1
```

Script Before Signing

Sample.ps1 - Notepad

File Edit Format View Help

```
Write-Host ‘Start of script’ -BackgroundColor Green
Write-Host ‘Display the % CPU Time utilization by the ISE’ -BackgroundColor Green
Get-Counter ‘\Process(powershell_ise)\% Processor Time’
```

Ln 1, Col 1

Script After Signing

Sample.ps1 - Notepad

File Edit Format View Help

```
Write-Host ‘Start of script’ -BackgroundColor Green
Write-Host ‘Display the % CPU Time utilization by the ISE’ -BackgroundColor Green
Get-Counter ‘\Process(powershell_ise)\% Processor Time’
# SIG # Begin signature block
# mIIEMwYJKoZIhvcNAQcCoIIEJDCCBCACAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# kjcCAQSgWsBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# agEAAsEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQU6vQAn5sf2qIxQqwWUDwTZnJj
# j5ufgfI9MIICOTCCAaagAwIBAgIQyLeyGZcGA4ZOGqK7VF45GDAJBgUrDgMCHQUA
# agEAAsEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQU6vQAn5sf2qIxQqwWUDwTZnJj
# kjcCAQSgWsBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# ...........
# SIG # End signature block
```

Ln 1, Col 1

Script signature block

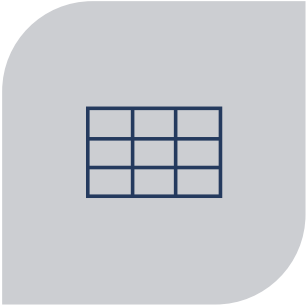
Hash Tables

Learnings covered in this Unit

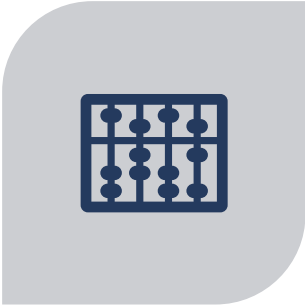
- What is a Hash Table
- Creating Hash Tables
- Working with Hash Tables
- Techniques and use cases

What is a Hash Table

Hash Table Overview

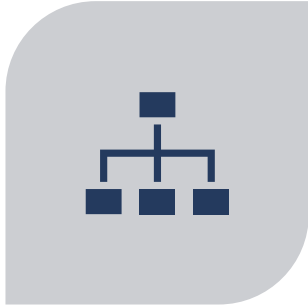


COLLECTION WITH A CONSISTENT SEARCH TIME



MEMORY LOCATION DETERMINED BY THE HASH ALGORITHM

USES KEY VALUE PAIRS TO STORE DATA



VALUE CAN BE ANY DATATYPE

Hash Table Storage

A collection where a hash function determines the memory location of the data stored

Input Data

Key: Mitchell
Value: 4,6692

Key: Douglas
Value: 42

Key: Euler
Value: 2,718

Key: Piwas
Value: 3,14

Hash Function

Key	Value	
		Memory
		Memory
Mitchell	4,6692	Memory
Piwas	3,14	Memory
Douglas	42	Memory
		Memory
Euler	2,718	Memory

Creating a Hash Table

Empty hash table

```
PS> $hash = @{}
```

Create and populate hash table

```
PS> $Server = @{
        'HV-SRV-1' = '192.168.1.1'
        Memory = 64GB
        Serial = 'THX1138'
    }

PS> $Server

Name                                   Value
----                                   -----
HV-SRV-1                               192.168.1.1
Serial                                 THX1138
Memory                                 68719476736
```

Creating a Hash Table from a string variable

```
PS> $string = "
Msg1 = Hello
Msg2 = Enter an email alias
Msg3 = Enter a username
Msg4 = Enter a domain name
"

PS> ConvertFrom-StringData -StringData $string

Name                                   Value
----                                   -----
Msg4                                   Enter a domain name
Msg3                                   Enter a username
Msg2                                   Enter an email alias
Msg1                                   Hello
```

Create a hash table using Group-Object

Group-Object outputs a **Key : value** pair.

Needs **-AsString** parameter to convert **key** to a string instead of an object.

```
PS> $svcshash = Get-Service |
Group-Object Status -AsHashTable -AsString

PS> $svcshash

Name        Value
----        -----
Stopped     {AeLookupSvc, ALG, AppMgmt...}
Running     {AppIDSvc, Appinfo...}

PS> $svcshash.Stopped

Status   Name                DisplayName
------   ----                -----------
Stopped  AeLookupSvc         Look up ser...
```

Accessing Hash Table Items

Accessing Hash Table Items

Access the item by key

Special characters allowed in key names

“Keys” and “values” properties available

Access Hash Tables Items - Examples

Display all items in hash table

```
PS> $Server

Name           Value
----           -----
HV-SRV-1       192.168.1.1
Serial         THX1138
Memory         68719476736
```

Return value using dot notation

```
PS> $Server.'HV-SRV-1'
192.168.1.1

PS> $Server.Serial
THX1138
```

Return value using “index” notation

```
PS> $Server["Serial"]
THX1138
```

Display All Hash Tables Keys and Values

Display all keys in hash table

```
PS> $Server.Keys
HV-SRV-1
Serial
Memory
```

Display all values in hash table

```
PS> $Server.Values
192.168.1.1
THX1138
68719476736
```

Note: Individual key lookup is fast, individual value lookup is slow on large tables

Modifying Hash Table Items

Adding Items To a Hash Table

Add or set key and value using index notation

```
PS> $Server["CPUCores"] = 4
```

Add or set key and value using dot notation

```
PS> $Server.Drives = "C", "D", "E"
```

Add key and value using hash table ADD method

```
PS> $Server.Add("HotFixCount", (Get-HotFix -Computer
$Server["HV-SRV-1"]).count)
```

Note: Adding a key that already exists will cause an error

Removing Items From a Hash Table

Remove key

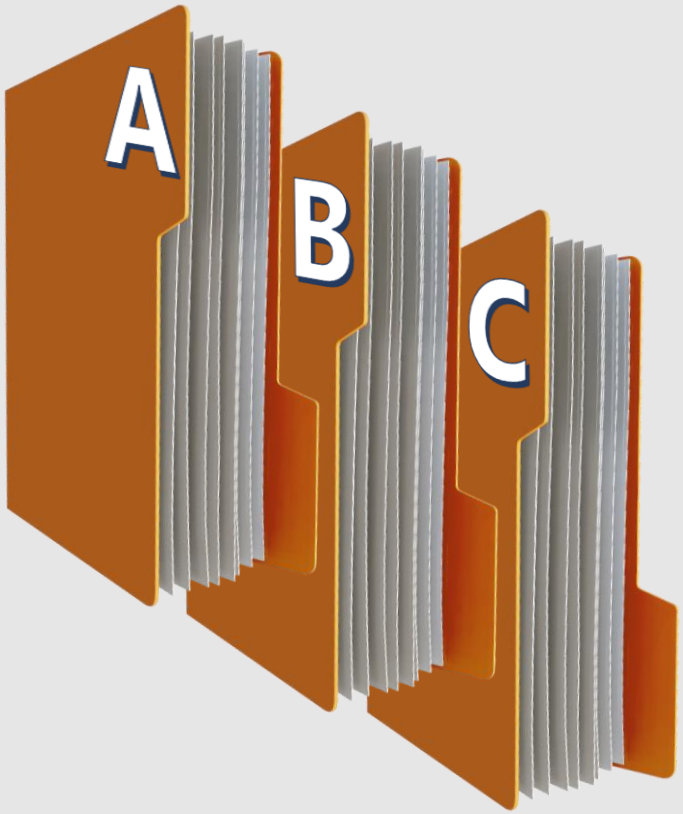
```
PS> $Server.Remove("HotFixCount")
```

Empty the Complete table

```
PS> $Server.Clear()
```

Sorting and Searching Hash Tables

Sorting Hash Tables



- Hash tables are intrinsically **unordered**
- It is **not** possible to sort a hash table as it's a **single** object
- **GetEnumerator()** reads the table one entry at a time, returning a **list** of objects on **key-value pairs**

Sorting Hash Tables - Example

```
PS> $Server.GetEnumerator() | Sort-Object -Property key

Name          Value
----          -----
CPUCores      4
Drives        {C, D, E}
HV-SRV-1      192.168.1.1
Memory        68719476736
```

Searching Inside Hash Tables

Searching on Key:

- Contains() or Containskey()
- Constant lookup time
- Case insensitive

Searching on Value:

- ContainsValue()
- Variable lookup time
- Case sensitive

Searching Hash Tables - Example

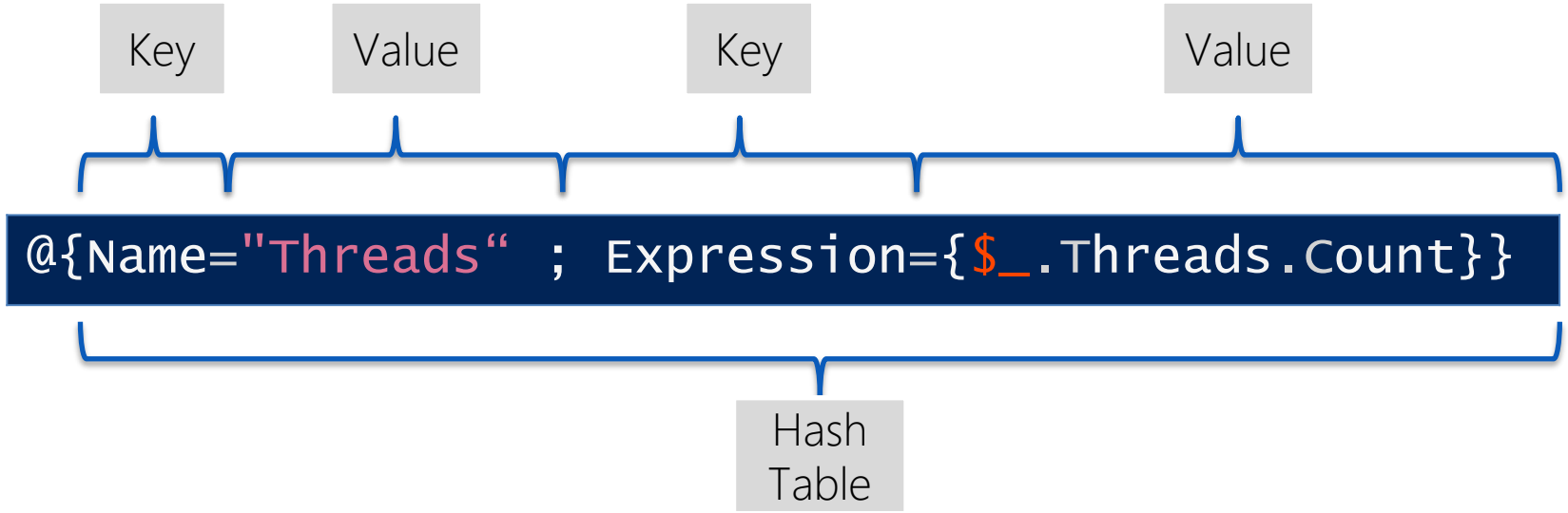
```
PS> $hash = @{"John"=23342;"Linda"=54345;"James"=65467}

PS> $hash.ContainsKey("Linda")     #Fast hashed key search
True

PS> $hash.ContainsValue(19)        #Slow non-hashed search
False

PS> $hash.ContainsValue(65467)
True
```

Hash Tables – Practical Use Cases

Calculated Properties – Simple Example

- Most display commands support **calculated** properties
- **Calculated** properties can use key:value pair of a hash table

```
PS> Get-Process | FT Name,@{Name = "Threads"; Expression = {$_.threads.count}}

Name                                                    Threads
----                                                    -------
aesm_service                                                  2
ApplicationFrameHost                                          5
Calculator                                                   28
```



Calculated Properties – Advanced Example

Using a code block in the expression key

```
PS> get-process | sort basepriority |
ft ProcessName,basepriority, @{
    name='priority'
    expression={
        switch ($_.basepriority) {
            {$_ -lt 8 }   {'low'}
            {$_ -eq 8 }   {'normal'}
            {$_ -gt 8 }   {'High'}
        }
    }
}

ProcessName          BasePriority priority
-----------          ------------ --------
Idle                            0 low
Spotify                         4 low
OUTLOOK                         8 normal
powershell_ise                  8 normal
services                        9 High
Spotify                        10 High
```

Code block

Hash Table

Calculated priority column added

Custom Object Creation

Use a hash table to create a PSObject that can be added to an array directly

```
$ping = Test-Connection -computername "dns.google“ -count 4
$pingmeasure = $ping |
Measure-Object -Property "ResponseTime“ -Maximum -Minimum -Average

$properties = @{
    'Name' = $object
    'pingtime' = $pingmeasure.average
    'pingcount' = $pingmeasure.count
    'pingMaxmum' = $pingmeasure.Maximum
    'pingMinimum' = $pingmeasure.Minimum
}

[array]$result += New-Object -TypeName PSObject -Properties
$properties
```

Custom Object Creation

Use a hash table to create a PSCustomObject

```
$CompInfo = [PSCustomObject]@{
    'ComputerName'    = $env:Computername
    'Domain'          = $env:USERDNSDOMAIN
    'DiskVolume'      = (Get-Volume | Where-Object -Property DriveLetter).DriveLetter
    'PhysicalMemory' = (Get-CimInstance -ClassName CIM_PhysicalMemory |
                            Measure-Object -Property Capacity -Sum).Sum / 1gb
}
```

```
PS C:\> $CompInfo


ComputerName Domain        DiskVolume PhysicalMemory
------------ ------        ---------- --------------
WIN10        CONTOSO.LOCAL          C             10
```

Splatting

A technique for passing arguments to commands

```
Get-ChildItem -Path c:\windows -File | Measure-Object -Average -Sum
-Maximum -Minimum -Property Length
```

Versus

```
$moparams = @{
    Average = $true
    Maximum = $true
    Sum = $true
    Minimum = $true
    Property = 'length'
}
$gciparams = @{
    Path = 'c:\windows'
    File = $true
}
Get-ChildItem @gciparams | Measure-Object @moparams
```

PowerShell Remoting Basics

Learnings covered in this Unit

Enabling PowerShell remoting

Understanding PowerShell remoting

Using PowerShell remoting

Enable PowerShell Remoting

Understanding PowerShell Remoting

PowerShell **Remoting** cmdlets allow for **code** to be executed on one or more **remote** machines

Modern remoting cmdlets use **CIM**, and **WS-MAN** with a dedicated **port**

Legacy remoting cmdlets use **DCOM** and **RPC**, with very **little** functionality

Requirements

Windows PowerShell **2.0** or later, on local **and** remote computers

Remoting must be **enabled** on client machines and is enabled by **default** on Windows Server 2012 and later server versions

Local Administrators or **Remote Management Users** are allowed access by default

Enable PowerShell remoting interactively

Enable-PSRemoting performs the following **actions**:

- Starts the **Windows Remote Management (WinRM)** service and sets it to **automatic** startup
- Creates an **HTTP listener** to accept remote requests on any IP address for TCP port **5985**
- Enables a **firewall exception** for WS-Management
- Several **other changes** occur as well, which can be found in the help documentation

Enable PowerShell Remoting using Group Policy

Set **WinRM** Service to **Automatic** Startup

Computer Configuration | Policies | Windows Settings | Security Settings | System Services | Windows Remote Management (WS-Management)

Set Windows **Firewall** Inbound **rule** for Windows Remote Management

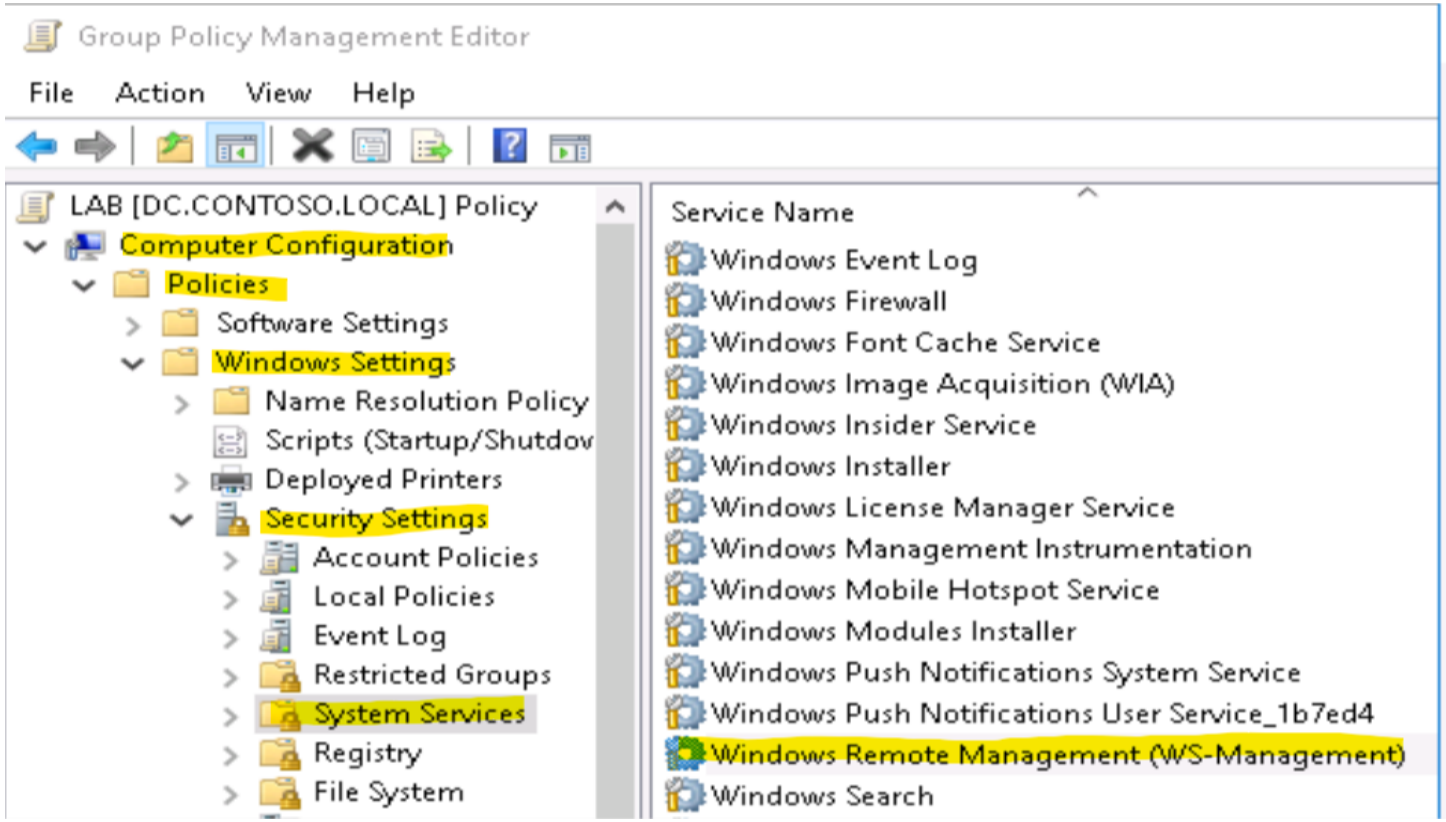
Computer Configuration | Policies | Windows Settings | Security Settings | Windows Firewall with Advanced Security | Inbound Rules | Windows Remote Management

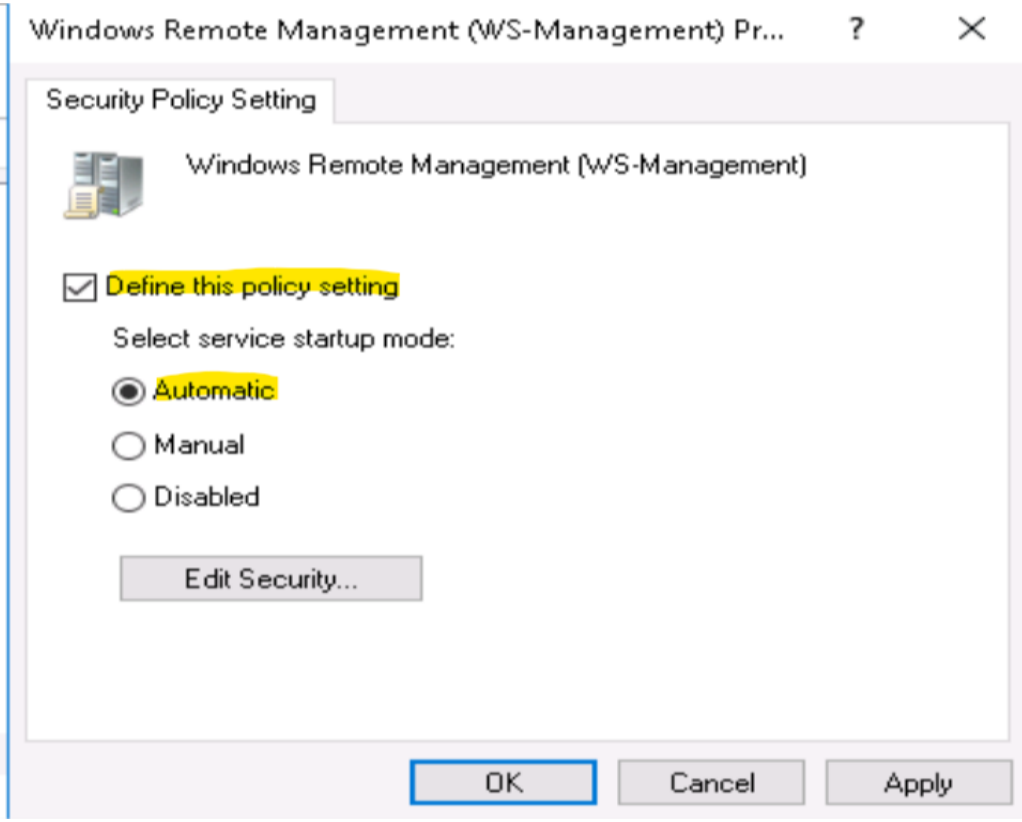
Allow remote server management (create **listeners**)

Computer Configuration | Administrative Templates | Windows Components | Windows Remote Management (WinRM) | WinRM Service | Allow remote server management through WinRM

Enable PowerShell Remoting using GPO – 1

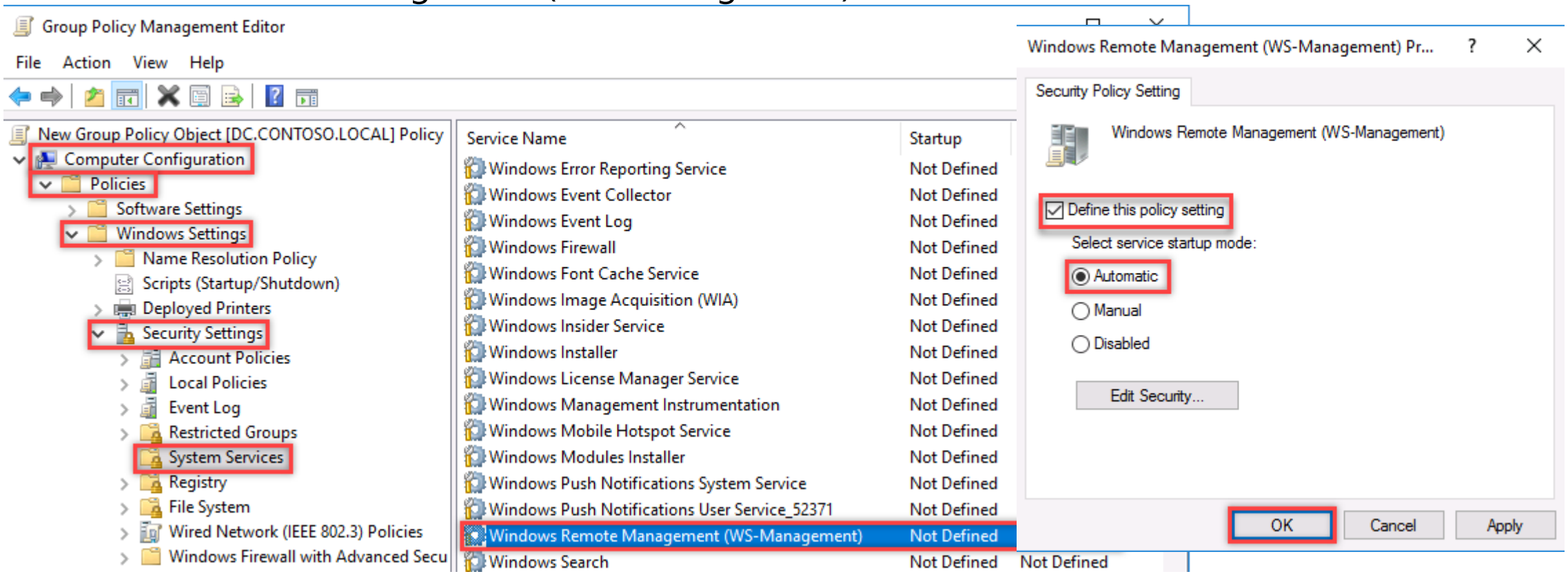
- Set WinRM Service to Automatic Startup
  - Computer Configuration | Policies | Windows Settings | Security Settings | System Services | Windows Remote Management (WS-Management)





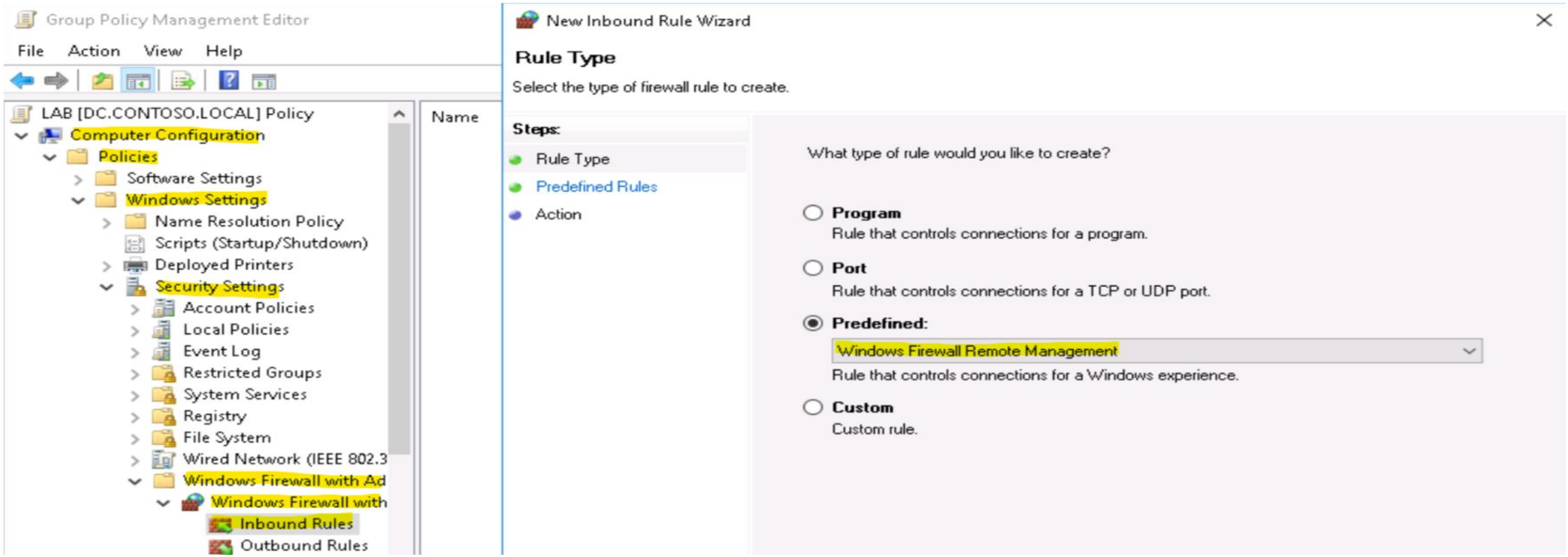
Enable PowerShell Remoting using GPO – 2

- Set WinRM Service to Automatic Startup
  - Computer Configuration | Policies | Windows Settings | Security Settings | System Services | Windows Remote Management (WS-Management)



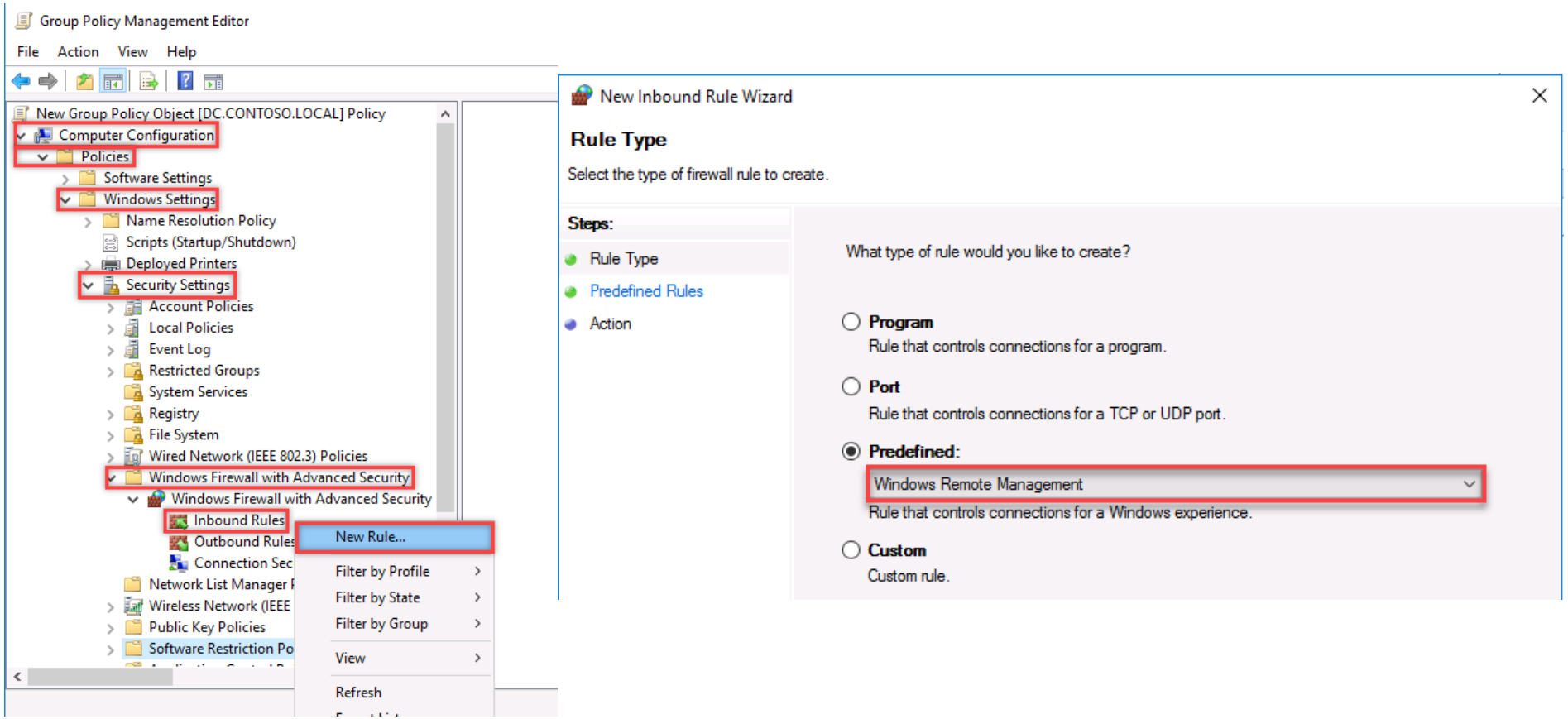
Enable PowerShell Remoting using GPO – 3

- Set Windows Firewall Inbound rule for Windows Remote Management
  - Computer Configuration | Policies | Windows Settings | Security Settings | Windows Firewall with Advanced Security | Inbound Rules | Windows Remote Management



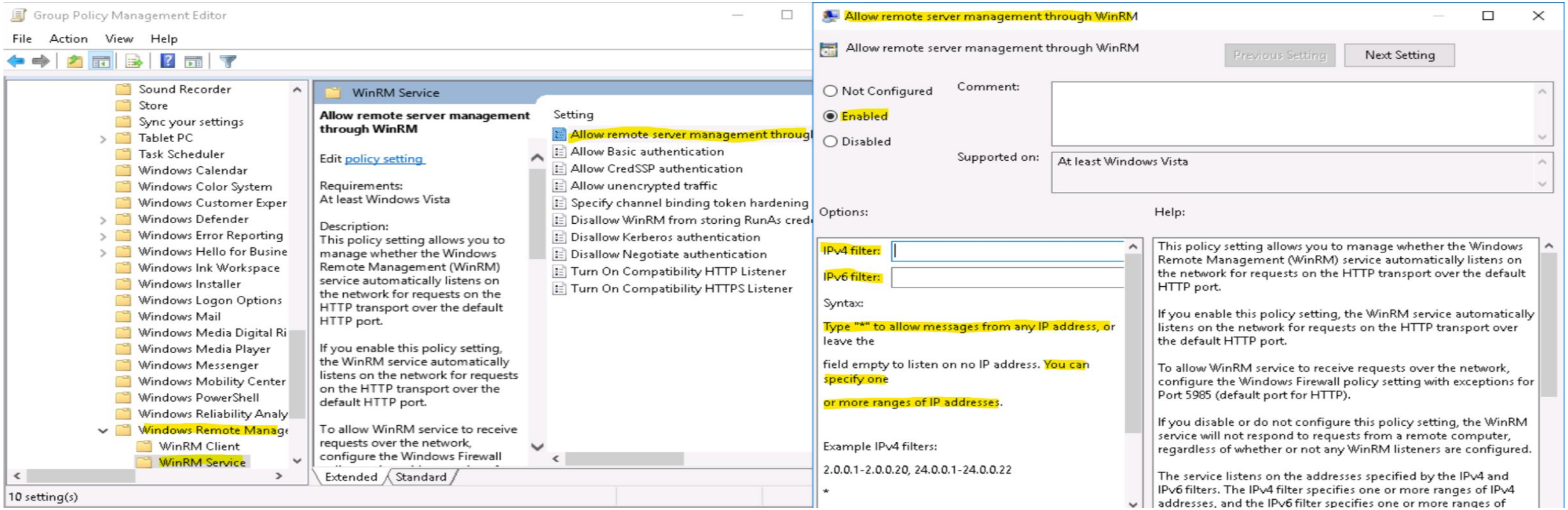
Enable PowerShell Remoting using GPO – 4

- Set Windows Firewall Inbound rule for Windows Remote Management
  - Computer Configuration | Policies | Windows Settings | Security Settings | Windows Firewall with Advanced Security | Inbound Rules



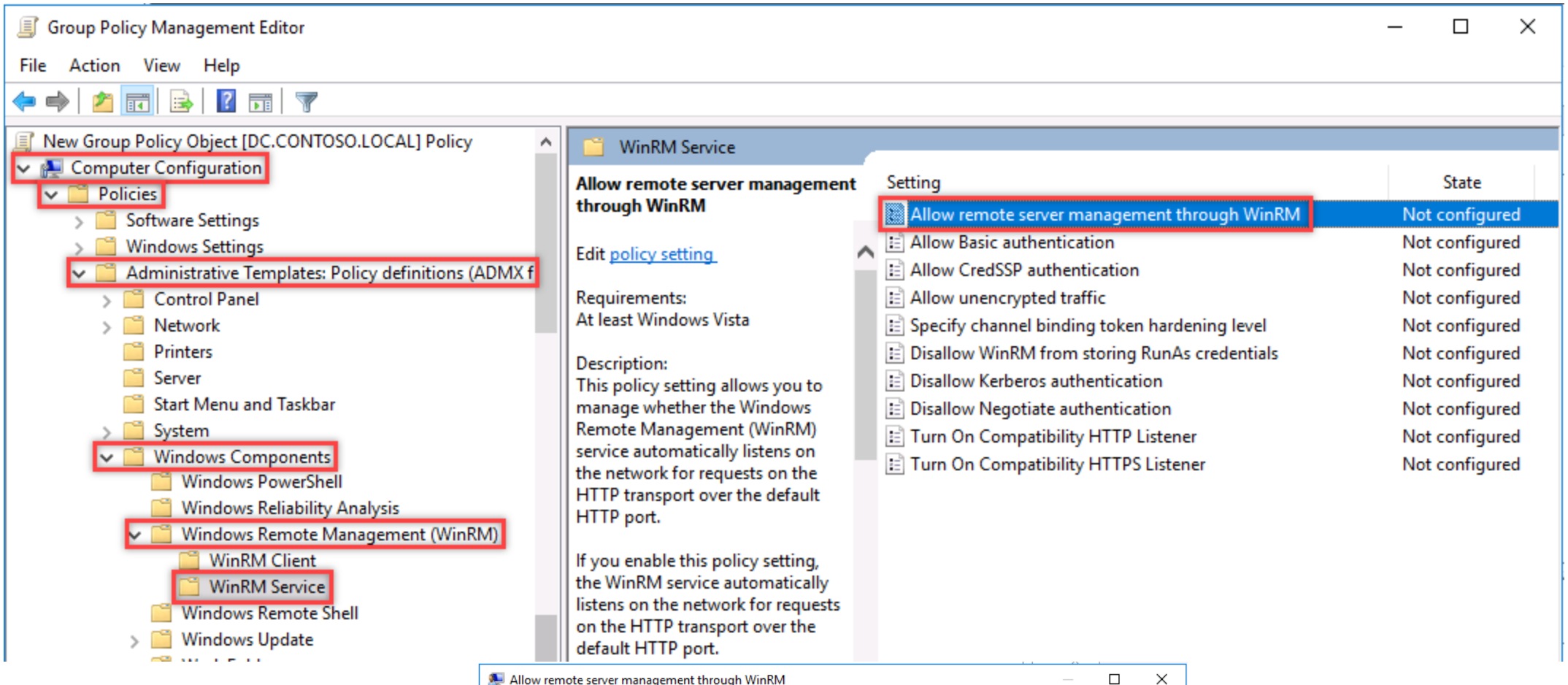
Enable PowerShell Remoting using GPO – 5

- Allow remote server management (create listeners)
  - Computer Configuration | Administrative Templates | Windows Components | Windows Remote Management (WinRM) | WinRM Service | Allow remote server management through WinRM



Enable PowerShell Remoting using GPO – 6

- Allow remote server management (create listeners)
  - Computer Configuration | Administrative Templates | Windows Components | Windows Remote Management (WinRM) | WinRM Service | Allow remote server management through WinRM



Using PowerShell Remoting

Types of Remoting

Native OS Remoting (Legacy)

- Built-In cmdlets that take a **ComputerName** parameter but do **not** contain a **session** parameter.
- Does **not** need PowerShell remoting **enabled**
- **Limited** functionality: uses **built-in** Windows services
- Transports over **DCOM** and **RPC**

```
PS> Get-Command -ParameterName ComputerName -Module Microsoft.PowerShell.Management
```

Remoting (Modern)

- **Requires** PowerShell remoting to be **enabled**
- **Sessions** are **created** on a single machine or group of machines
- Transports over **WS-MAN** on a dedicated **port**

Native OS Remoting

- Typically, Windows resource commands that accomplish one thing per cmdlet
- Uses built-in Windows services
- Transports using RPC
- Target machines do not need PowerShell remoting enabled
- Works on all Windows operating systems without any special configuration.

```
PS> Get-Service -Computername DC

Status   Name               DisplayName
------   ----               -----------
Stopped  AarSvc_203ac40f    Agent Activation Runtime_203ac40f
Running  AdobeARMservice    Adobe Acrobat Update Service
Running  AdobeUpdateService AdobeUpdateService
...
```

PowerShell Remoting

Operates using WS-Man (**WinRM**) using **a** dedicated **port**

Flexible and **powerful**, allowing any **command** on any **machine**

Can execute commands on **multiple machines** at once

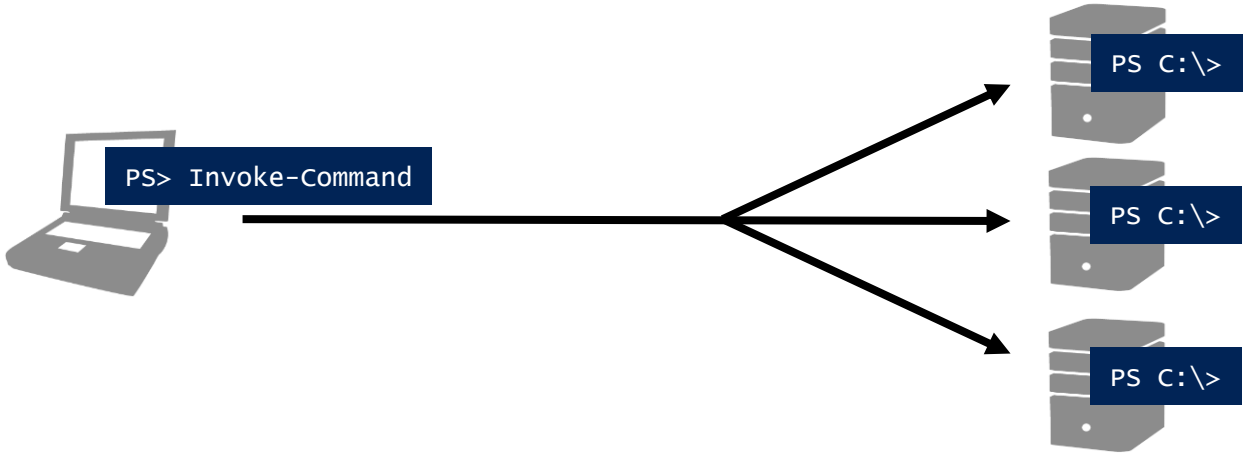
Supports **interactive** sessions, similar to **SSH** sessions

Invoke-Command

Execute any **script block** on any number of machines

Returns the results with a **PSComputerName** property

Execution happens in **parallel**



Remote Computers

Script block to be executed

```
PS> Invoke-Command -ComputerName MS,DC,Win10 -ScriptBlock {Get-Culture}

LCID        Name        DisplayName                    PSComputerName
----        ----        -----------                    --------------
1033        en-US       English (United States)        MS
1033        en-US       English (United States)        DC
1033        en-US       English (United States)        WIN10
```

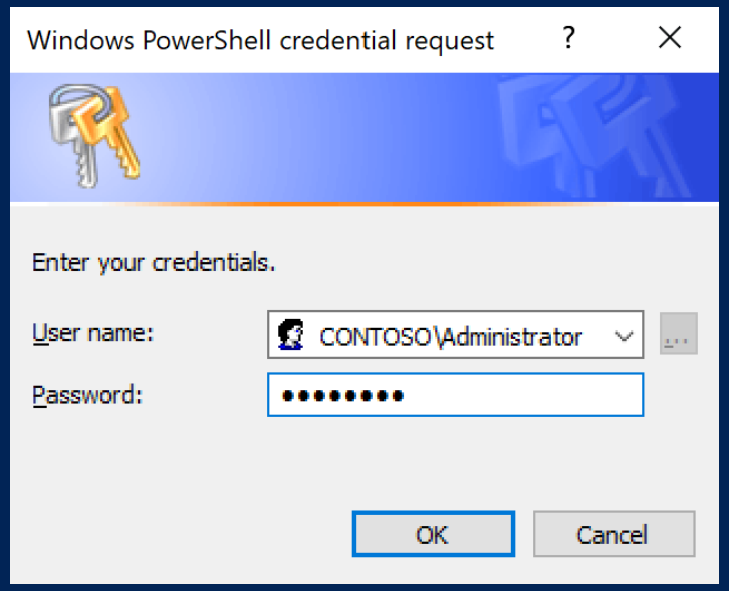
Using Alternate Credentials

The **-Credential** parameter specifies alternate credentials to **authenticate** to the remote machine

Credentials can be saved to a **variable** with **Get-Credential**

Useful when logged in as a standard **user** account but **administrative permissions** are required on the **remote** machine

```
PS> Invoke-Command -ComputerName MS -ScriptBlock {$env:USERNAME}
    -Credential CONTOSO\Administrator

Administrator
```



Invoke-Command with Script Files

The **-FilePath** parameter sends a **local** script to a **remote** computer

Converts code from file into a **script block**

Use **-ArgumentList** to specify the values of **parameters** for the **script**

```
PS> Invoke-Command -ComputerName MS, DC -FilePath C:\MyScript.ps1

LCID        Name        DisplayName                    PSComputerName
----        ----        -----------                    --------------
1033        en-US       English (United States)        MS
1033        en-US       English (United States)        DC
```

Temporary vs Persistent Sessions

Temporary sessions

- **Closes** the session when the command is **completed** or when the interactive session **ends**
- Uses the **-ComputerName** parameter
- Executes a **single** script block on a remote machine

Persistent sessions

- A **PSSession** that remains **available** even after a command is **completed** or an interactive session ends
- Uses the **-Session** parameter
- Can **disconnect** and **reconnect**, as needed
- Remains open until it is **deleted** or **times out**

Using Persistent Sessions

New-PSSession creates a persistent connection to a remote computer

Can be used with **Invoke-Command**, ***-PSSession** cmdlets, and other cmdlets

Generally saved into a **variable** for easier reuse

Session **exists** on the **remote** computer and is **available** to **connect** to and use as needed

```
PS C:\> $Session = New-PSSession -ComputerName MS

PS C:\> Invoke-Command -Session $Session -ScriptBlock {$Var = 123}

PS C:\> Invoke-Command -Session $Session -ScriptBlock {$Var}
123
```

Variable still exists on remote machine