Windows PowerShell Scripting and Toolmaking Lab Guide

Welcome to the Windows PowerShell Scripting and Toolmaking Lab Guide!

In this lab manual, you will find that most of the lab steps are intended to be general and will require you to do research and discover how to perform many tasks by yourself. Of course, most of the labs are based upon the lecture and demonstrations that your instructor performed before you start each section, so you should have a good foundation before you begin the lab work.

Some of you may have a bit of a harder time with these theme-based labs. For that reason, you will see many "Notes", "Hints" and "Spoiler Alerts" scattered throughout the labs. If you want less hand-holding, ignore these callouts and try to do things on your own as much as you can. When you need more assistance, these extra bits of detail can help get you on the right track.

If you get stuck anywhere in the labs, there are sample answers to all of the projects. These files are located in the C:\LabFiles\Answers folder within your virtual machines. Each module and lab will have its own folder with the files representing the solutions to each project. Only use these files as a last resort.

Of course, you can always ask your instructors for help as well. They will be happy to assist you and steer you in the right direction!

Note:
During the labs, after your VMs have been running for a day or so, you may see an "Activate Windows" message on the desktop of the VMs. If this happens, you may dismiss the message by restarting the VM. If the message reappears, try running the following commands from an Elevated PowerShell Prompt:

```
SLMGR -ReArm

Restart-Computer
```

This should cause the message to disappear for 10 days.

Module 1

Lab A: Using Windows Powershell

(30 Minutes)

In this lab, you will review basic PowerShell commands and techniques. The lab steps in this first module will be far more detailed than the labs in the remaining modules. This module is intended to give you a jump-start in case you are not as familiar with basic PowerShell operations. For most of you, this should be a review that will solidify your existing basic knowledge of the PowerShell environment.

Exercise 1: Exploring Windows PowerShell

Task 1: Use Windows PowerShell

Windows PowerShell is not the command prompt. Some commands have been replicated for convenience, but are not really the same. In this task, you will use some of the basic commands within Windows PowerShell.

1. Log on to the Echo VM as Speck with a password of Pa$$w0rd.
2. Right-click the PowerShell icon on the taskbar and select Run as Administrator, then click Yes.

Note:
If you do not have a shortcut to PowerShell on the taskbar, click Start, and Type PowerShell, then right-click the PowerShell icon and select Run as Administrator.

3. Try using standard command-line tools, like: (Type each command and then press ENTER.)

```
Cd

dir

net share

hostname

# Try other commands that you know from the command prompt.
```

4. Type the following command and then press ENTER:

```
help dir
```

5. What command is the help output for?

   _____

6. Type the following commands: (Type each command and then press ENTER.)

```
Help

Get-Command | More
```

7. Page through the output with the spacebar. Press CTRL+C after paging a few times.

8. The output of **get-command** is very lengthy and difficult to read. There is a built-in tool called Out-GridView that can display the information in a more user-friendly fashion. (Enter the following on the same line and then press ENTER.)

```
Get-Command | Sort-Object Name | Out-GridView
```

9. Once you have found a command you are interested in, you may want additional help for the command. Type the command below followed by the Enter key:

```
Help Get-Service -Full
```

10. Page through the output by pressing the space bar. Notice the examples of usage near the end.
11. Type the following directory commands: (Type each command and then press ENTER.)

```
dir C:\

dir C:\ | format-list
```

Task 2: Using the PowerShell Pipeline

One of the most fundamental and powerful concepts in PowerShell is the Pipeline. This is the process of sending the output of one command to another for further processing. PowerShell takes this capability to new heights.

In this task you will experiment with various pipeline techniques.

12. In the PowerShell prompt, type the commands below, followed by the Enter Key: (Some of the commands may wrap in the examples below and should be typed as a continuous command before pressing Enter.)

```
Get-Service

Get-Service | Sort-Object Status

Get-Service | Where-Object DisplayName -like "*windows*" | Sort-
Object Status,name

Get-Service | Where-Object DisplayName -like "*windows*" | Sort-
Object Status,name | FT DisplayName,Status,Name -Auto

New-Item C:\Reports -ItemType Directory

Set-Location C:\Reports

Get-Service | Where-Object DisplayName -like "*windows*" | Sort-
Object Status,name | FT DisplayName,Status,Name -Auto | Out-File
C:\Reports\WindowsServices.txt

Get-ChildItem

Notepad C:\Reports\WindowsServices.txt
```

13. Minimize Notepad and return to PowerShell.
14. How many pipeline symbols did you use in the final Get-Service command?

15. The next commands are a more complex example of using the pipeline to send an object of a specific type from one command to another in the pipeline. Type the commands below, followed by the Enter Key:

```
Notepad;Notepad;Notepad
# You should now have 4 notepad instances running.

Get-Process

Get-Process Notepad

Get-Process Notepad | Stop-Process
```

16. Now, let's find out how Stop-Process knew to stop the Notepad processes in particular. Use the following command to find out what kind of object is PRODUCED by Get-Process: (Look at the top of the output for the TypeName:.)

```
Get-Process | Get-Member
```

17. Now, run the following command to find out what Parameters can be CONSUMED by Stop-Process. (In the PARAMETERS section, look for the -InputObject parameter and you will see the type of object it can consume.)

```
Help Stop-Process -Full
```

Task 3: PowerShell Objects and Properties

Another important concept in PowerShell is the fact that most of the data produced by PowerShell consists of Objects and their associated Properties and Methods. Properties are details ABOUT the objects, while Methods are things you can DO to objects.

In this task, you will explore various types of objects and the properties they contain.

18. Type the commands below followed by the ENTER key:

```
Get-Service

Get-Service *win* | Format-List *

Get-Service | Get-Member

Get-Process *win* | Format-List *

Get-Process | Get-Member

Get-AdUser -Filter * -Properties * | Get-Member
```

19. Now, let's examine the object nature of PowerShell by placing the objects into a variable. Type the commands below followed by the Enter key:

```
Notepad

Get-Process

$MyProcs = Get-Process

$MyProcs

$MyProcs | Where-Object Name -like Notepad | Format-List *

Stop-Process -Name Notepad

$MyProcs

Get-Process

Notepad

$MyNotepad = Get-Process Notepad

$MyNotepad | Get-Member

$MyNotepad | Format-List *

$MyNotepad.Name

$MyNotepad.MainModule

$MyNotepad.Kill()
```

Lab B: Using Variables

(30 Minutes)

In this lab, you will review the usage of PowerShell Variables.

Exercise 1: Creating Variables and Interacting with Them

Task 1: Create and work with variables and variable types

20. Log on to the Echo VM as Speck with a password of Pa$$w0rd.
21. Open the PowerShell prompt as Administrator.
22. Type the following commands and press ENTER:

```
$myValue = 1

$myValue

$myValue = 'Hello'

$myValue

Get-Help Get-Variable -Full

Get-Variable myValue | fl *

Get-Variable myValue -ValueOnly

[int]$myValue = 1

$myValue

$myValue = 'Hello'
```

Task 2: View all variables and their values

23. In the PowerShell window, type the following command and press ENTER:

```
Get-Variable
```

Task 3: Remove a variable

24. In the PowerShell window, type the following commands and press ENTER:

```
Get-Help Remove-Variable -Full

Remove-Variable myValue
```

Exercise 2: Understanding Arrays and Hashtables

Task 1: Create an array of services

25. In the PowerShell window, type the following commands and press ENTER:

```
get-help about_arrays

$services = Get-Service

$services
```

```
$services.GetType().FullName
```

Task 2: Access individual items and groups of items in an array

26. In the PowerShell window, type the following commands and press ENTER:

```
$services[0]

$services[1]

$services[-1]

$services[0..9]
```

Task 3: Retrieve the count of objects in an array

27. In the PowerShell window, type the following command and press ENTER:

```
$services.Count
```

Task 4: Create a hashtable of services

28. In the PowerShell window, type the following commands and press ENTER:

```
get-help about_hash_tables

$serviceTable = @{}
```

```
foreach ($service in $services) {
   $serviceTable[$service.Name] = $service
}

$serviceTable
```

Task 5: Access individual items in a hashtable

29. In the PowerShell window, type the following commands and press ENTER:

```
$serviceTable['BITS']

$serviceTable['wuauserv']

$serviceTable | gm

$serviceTable.Count
```

Task 6: Enumerate key and value collections of a hashtable

30. In the PowerShell window, type the following commands and press ENTER:

```
$serviceTable.Keys

$serviceTable.Values
```

Task 7: Remove a value from a hashtable

31. In the PowerShell window, type the following commands and press ENTER:

```
$serviceTable.Remove('BITS')

$serviceTable['BITS']
```

Exercise 3: Using Single-Quoted and Double-Quoted Strings and the Backtick

Task 1: Learn the differences between single-quoted and double-quoted strings

32. In the PowerShell window, type the following commands and press ENTER:

```
$myValue = 'Hello'

"The value is $myValue"

'The value is not $myValue'
```

Task 2: Use a backtick (`) to escape characters in a string

33. In the PowerShell window, type the following commands and press ENTER:

```
"Nor is the value `$myValue"
```

```
dir 'C:\Program Files'

dir C:\Program` Files
```

Task 3: Use a backtick (`) as a line continuance character in a command

34. In the PowerShell window, type the following command and press ENTER:

```
Get-Service `
-name "Windows Update"
```

Exercise 4: Using Arrays

Task 1: Learn how to extract items from an array

35. In the PowerShell window, type the following command as a single line and press ENTER:

```
$days =
@('Sunday','Monday','Tuesday','Wednesday','Thursday','Friday','S
aturday')
```

36. In the PowerShell window, type the following command and press ENTER:

```
$days[2]
```

Task 2: Discover the properties and methods for an array

37. In the PowerShell window, type the following commands and press ENTER:

```
$days | Get-Member

Get-Member -InputObject $days
```

Exercise 5: Using Contains, Like and Equals Operators

Task 1: Use contains, like and equals operators to find elements in arrays

38. In the PowerShell window, type the following command and press ENTER:

```
get-help about_comparison_operators
```

39. In the PowerShell window, type the following command as a single line and press ENTER:

```
$colors =
@('Red','Orange','Yellow','Green','Black','Blue','Gray','Purple'
,'Brown','Blue')
```

40. In the PowerShell window, type the following command and press ENTER:

```
$colors -like '*e'

$colors -like 'b*'

$colors -like '*a*'

$colors -contains 'White'

$colors -contains 'Green'

$colors -eq 'Blue'
```

Task 2: Understand the difference when using those operators with strings

41. In the PowerShell window, type the following commands and press ENTER:

```
$favoriteColor = 'Green'

$favoriteColor -like '*e*'

$favoriteColor -contains 'e'
```

Lab C: *Bonus* Writing Basic Scripts

This is a bonus lab that you may complete if time permits.

Exercise 1: Writing Scripts and Configuring the Execution Policy

In this exercise, you will configure the PowerShell Execution Policy and write a few basic PowerShell scripts.

Task 1: Write Windows PowerShell Scripts

Entering scripts more than a few lines long inside Windows PowerShell is not a very convenient way to perform complex or repetitive tasks. For that reason, it is helpful to save the script to a file and run it later. Running scripts is disabled by default, so the capability must be enabled.

In this task, you will write scripts for Windows PowerShell.

42. Log on to Echo as Speck with a password of Pa$$w0rd.
43. Open Windows PowerShell as Administrator.
44. Type the following command and then press ENTER. (This will enable the ability to run scripts.)

```
Set-ExecutionPolicy RemoteSigned
```

45. Type the following commands: (Type each line and then press ENTER.)

```
md c:\scripts

cd c:\scripts

notepad mydir.ps1          # (Click Yes to create the file.)
```

Note:
No, we should not be using Notepad to edit our PowerShell scripts. We are using this demonstrate WHY you should use PowerShell ISE instead. That is coming shortly in another task.

46. In the Notepad window, type the following lines of code. (Notice the new line we have added at the top of the script to prompt for input.) (Type each line and then press ENTER.)

```
$Folder = read-host "Enter a path to a folder"
$Files = get-childitem $Folder | select-object mode, name
foreach ($Item in $Files)
{
  If ($Item.mode -eq "d-----")
  {write-host $Item.name -foregroundcolor "green"}
  else
  {write-host $Item.name}
}
```

47. Save the file and then minimize Notepad.
48. In Windows PowerShell, type the command `c:\scripts\mydir.ps1` and then press ENTER. (You will be prompted for a folder.)
49. Type `c:\` and then press ENTER.
50. Run the script again, but this time when prompted for a folder, type `c:\windows` and then press ENTER.

Note:
You will find example scripts for the lab and many other samples in the C:\LabFiles folder.

Task 2: Use Windows PowerShell ISE

Windows PowerShell ISE provides a built-in Integrated Scripting Environment. This provides additional help via a colorized code-editor with code completion and intellisense.

In this task, you will use Windows PowerShell ISE to write scripts.

51. Right-click the PowerShell icon on the taskbar and select Run ISE as Administrator, then click Yes.
52. Click **Windows PowerShell ISE**.
53. Maximize Windows PowerShell ISE.
54. Notice the list of commands in the right window pane.
55. Click the down-arrow next to Script at the top of the blue window.
56. In the top window, type the following lines of code: (Type each line and then press ENTER.)

Note:
The first line in the following script wraps in this document due to formatting. You should type it as one continuous line in Windows PowerShell ISE.

```
$getinput = read-host "Enter a list of computer names separated
by commas and no spaces"
$hostnames = $getinput.split(",")
foreach ($thishost in $hostnames)
{
   $result = Test-Connection $thishost -Count 1 -Quiet
   If ($result)
   {Write-Host "$thishost Succeeded!" -ForeGroundColor Green}
   Else
   {Write-Host "$thishost Failed!" -ForeGroundColor Red}
}
```

57. Click the Run Script icon (it looks like a “Play” button) in the icon bar of Windows PowerShell ISE.
58. In the blue PowerShell window at the bottom, when the input prompt appears, type Alpha,Bravo,Echo,NoPC and then press Enter (Do not put any spaces after the commas.)
59. Scroll through the script results in the blue window.
60. Save the script to a new folder called **C:\Scripts** as **PingMany.ps1**.
61. From the taskbar, switch to or open a new PowerShell prompt and type the following and press Enter:

```
cd \scripts

.\PingMany.ps1
```

62. When the input prompt appears, type Alpha,Bravo,Echo,NoPC and then press ENTER.

Lab D: *Bonus* Remote Management

This is a bonus lab that you may complete if time permits.

Exercise 1: PowerShell Remote Management

In this exercise, you will explore the options available to remotely manage systems with PowerShell.

Task 1: Standard PowerShell Cmdlet Remote Management

Many Cmdlets already have a remote management capability built-in. These commands generally use RPC communications to get the job done.

63. Log on to the Echo VM as Speck with a password of Pa$$w0rd.

64. Open the Start screen and type PowerShell, then right-click the Windows PowerShell result and select Run as Administrator then click Yes.
65. Type the commands below followed by the Enter key: (Some of the commands may wrap in the examples below. Type each command as a single contiguous command, then press ENTER.)

```
Get-Process -ComputerName Alpha

Get-Process -ComputerName Alpha,Bravo | Sort MachineName | FT -
GroupBy MachineName

Get-EventLog Security -Newest 10 -ComputerName Alpha,Bravo |
Sort MachineName | FT -GroupBy MachineName

Get-AdComputer -Filter *

Get-Service *win* -ComputerName (Get-AdComputer -F * | ForEach
Name) | Sort MachineName | FT -GroupBy MachineName

Get-WindowsFeature -ComputerName Alpha | More

Add-WindowsFeature Telnet-Client -ComputerName Alpha
```

66. Try the following commands:

```
Get-Volume

Get-Volume -ComputerName Alpha
```

67. Did the Second command work?

Task 2: Enabling PowerShell Remote Management with Group Policy

PowerShell Remoting is already enabled by default on Windows Server 2012, 2012 R2 and 2016. Windows clients must have PowerShell Remoting Enabled for it to work properly. To enable PowerShell Remoting on Windows clients and older verions of Windows server, you can either run the command manually, or use Group Policy.

In this task, you will enable PowerShell Remoting through a GPO.

68. Log on to the Echo VM as Speck with a password of Pa$$w0rd.
69. Open the Start menu and type gpmc.msc, then press Enter.
70. Navigate to Forest: hq.local, Domains, hq.local.
71. Right-click hq.local and select Create a GPO in this domain and link it here.
72. Type WinRM Settings and click OK.
73. Right-click WinRM Settings and select Edit.
74. Expand Computer Configuration, Windows Settings, Security Settings, System Services.
75. Open the Windows Remote Management service.
76. Select Define this policy setting, then select Automatic and click OK.
77. Expand Computer Configuration, Windows Settings, Security Settings, Windows Firewall with Advanced Security, Windows Firewall with Advanced Security, and open Inbound Rules.
78. Right-click Inbound Rules and select New Rule.
79. Select Predefined and choose Windows Remote Management from the list.
80. Click Next, Next and then Finish.
81. Expand Computer Configuration, Administrative Templates, Windows Components, Windows Remote Management, WinRM Service.
82. Open Allow remote server management through WinRM.
83. Select Enabled.
84. In the Options section type * for the IPv4 and IPv6 filters, then click OK.

85. Close the Group Policy Management Editor and the Group Policy Management Console.
86. Open the Start menu and type PowerShell, then right-click the Windows PowerShell result and select Run as Administrator then click Yes.
87. Type **`gpupdate`** and press Enter.

Task 3: Using PowerShell Remoting for Remote Management

In some cases the built-in cmdlets do not have remote capabilities. Also, the commands that do have the capability use RPCs which can be problematic when communicating through firewalls. Finally, there are sometimes occasions when you want the processing of a command to be performed by the remote computer before the results are returned.

To provide better remote management, Microsoft introduced PowerShell Remoting with PowerShell 2.0. It relies on WinRM for remote connectivity which is more secure and uses only one port for communications.

In this task, you will explore the PowerShell Remoting options.

88. Log on to the Echo VM as Speck with a password of Pa$$w0rd.
89. Open the Start menu and type PowerShell, then right-click the Windows PowerShell result and select Run as Administrator then click Yes.
90. Type the commands below, followed by the Enter key: (Some of the commands may wrap in the examples below. Type each command as a single contiguous command, then press ENTER.)

```
Invoke-Command -ComputerName Alpha {Get-Volume}

Invoke-Command -ComputerName Alpha,Bravo {Get-Volume} | FT
DriveLetter,DriveType,FileSystem,@{L="Size(GB)";E={"{0:N2}" -F
($_.Size/1GB)}; A="Right"} -Auto -GroupBy PSComputerName

Add-WindowsFeature -ComputerName Alpha,Bravo Telnet-Server
```

```
# The above command fails because Add-WindowsFeature cannot
# accept multiple computer names

Invoke-Command -ComputerName Alpha,Bravo {Add-WindowsFeature
Telnet-Server}
```

91. To open a PowerShell prompt against a remote computer, type the following:

```
Enter-PSSession Alpha
```

92. You should see the name of the computer you are connected to in front of the PS Prompt. Now type the following:

```
Hostname

IPConfig.exe

Get-NetIPAddress

Get-Process

Get-Service

Exit
```

Module 2

Lab A: Creating a Parameterized Script

(30 Minutes)

Beginning with this lab, and proceeding for several upcoming modules, you will be building a complete tool. That means your work in this module will be used in the next module, and the next, and the next, and so on. We've provided you with three complete labs to work on, because repetition is an important aspect of learning and reinforcement. However, each lab has a slightly different twist, although that may not become obvious until later modules.
Try to complete as many of these labs as possible. If you run out of time before completing all three, that's fine. If you finish up a future lab early, you can always come back to earlier ones. Also, we've provided you with a Starting Point for each lab. That way, even if you get completely stuck and run out of time, you'll be able to work on the next lab using our Starting Point.

Exercise 1

1. Log on to the Echo VM as Speck with a password of Pa$$w0rd.
2. Run a PowerShell prompt as Administrator.
3. Create a folder named C:\Scripts using the following commands:

```
md C:\Scripts

cd C:\Scripts
```

4. Run PowerShell ISE as Administrator.
5. Create a script named C:\Scripts\Get-OSInfo.ps1. That script should parameterize the following command:

```
Get-WmiObject -Class Win32_OperatingSystem -ComputerName ECHO |
  Select-Object Version,ServicePackMajorVersion,BuildNumber,
                OSArchitecture
```

Hint:
Add a Param block at the top of the script, then add a $ComputerName variable inside the Param block. Use the $ComputerName variable instead of ECHO.
A valid Param block could look like this:

```
Param (
        $Param1,
        $Param2,
        $Param3
     )
```

6. Run the script to test it.

Exercise 2

7. Create a script named C:\Scripts\Get-DiskInfo.ps1. The Get-DiskInfo script should parameterize the following command:

```
Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -
Computer ECHO |
  Where-Object { $_.FreeSpace / $_.Size * 100 -lt 99 }
```

Hint:
Add a Param block at the top of the script, then add the $ComputerName, $DriveType and $PercentFree variables inside the Param block. Use the $ComputerName variable instead of ECHO, use the $DriveType variable in place of the number 3 and use the $PercentFree variable in place of the number 99.

Also, the Pipe (|) symbol is a natural breakpoint. Use a hard carriage-return after the pipe symbol, then indent the continuation of the command below it. This makes your code more legible.

8. This command will display all local disks having free space of less than 99%. Notice that there are THREE pieces of information that should be parameterized:
    - The computer name
    - The drive type filter
    - The amount of free space needed to pass the threshold for output

9. Also notice that this function will not always return any output, even if it is running correctly. Make sure you understand why.

Exercise 3

10. Create a script named C:\Scripts\Invoke-OSShutdown. The Invoke-OSShutdown script should parameterize the following command:

```
Get-WmiObject –Class Win32_OperatingSystem –ComputerName BRAVO |
  Invoke-WmiMethod –Name Win32Shutdown –Arg 4 |
  Out-Null
```

Hint:
Add a Param block at the top of the script, then add the $ComputerName $Arg variables inside the Param block. Use the $ComputerName variable instead of ECHO and use the $Arg variable in place of the number 4.

11. There are two pieces of information that you will need to parameterize – and one of them is the **4** being passed to –Arg. For now, your function can simply accept a number, and it does not need to check to ensure that the number is valid. For your information (and so that you can test the script), valid values are:
    0 - Log Off (4 for forced log off)
    1 – Shutdown (5 for forced shutdown)
    2 – Restart (6 for forced restart)
    8 – Power Off (12 for forced power off)
    "Forced" means that applications are not allowed to cancel the operation, which means unsaved data may be lost. At this time, your function does not need to check to make sure the input is valid. That will come later.

Note:
Notice the use of Out-Null in the command that you have been given. Normally, Invoke-WmiMethod will produce a WMI output object with a "ReturnValue" property. That property contains a numeric code, with 0 indicating success and nonzero indicating an error. For now, we wish to merely suppress that output, and piping it to Out-Null will do so.

Win32_Shutdown will return an error if you try to log off of a computer where you are not logged on. This is expected; you can log on to all of your virtual machine computers if you want to see the command run without error.

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module02\LabA

Lab B: Creating a Parameterized Function

(15 minutes)

If you did not have time to complete the previous lab, you can find Starting Point scripts in your lab files under C:\LabFiles\StartingPoint\Module02\LabB. Copy the files in this folder into C:\Scripts before you begin.

Exercise 1

1. In this lab, you will combine your three scripts into a single script:
   - C:\Scripts\Get-OSInfo.ps1
   - C:\Scripts\Get-DiskInfo.ps1
   - C:\Scripts\Invoke-OSShutdown.ps1
2. Create a new folder called: C:\Scripts\oldWork and move your three original scripts into that folder.
3. Run PowerShell ISE as Administrator.
4. Open the three scripts from your C:\Scripts\oldWork folder.
5. Make each of those three scripts into functions named Get-OSInfo, Get-DiskInfo, and Invoke-OSShutdown.
6. Put the three functions into a script named C:\Scripts\MyTools.ps1.

Hint:
To convert a script into a function, wrap the entire script in a function body. Following is an example of a function declaration:

```
function TestOne {
     # My Script code goes here…
}
```

7. Inside PowerShell ISE, highlight all of the functions in the MyTools.ps1 file and press the F8 key (Run Selection).
8. In the blue PowerShell prompt window at the bottom of PowerShell ISE, run the following commands:

```
Get-OSInfo –ComputerName ECHO

Get-DiskInfo –ComputerName ECHO –DriveType 3
```

9. You will need to modify the parameters names if you did not use –ComputerName and –DriveType as your parameter names. If you didn't use those parameter names… just out of curiosity… why not?

Spoiler Alert!
If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module02\LabB

Module 3

Lab A: Creating and Testing a Manifest Module

(30 minutes)

In this lab, you will work with the C:\Scripts\MyTools.ps1 script from the last lab in the previous module.
If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab:

- In File Explorer, open C:\LabFiles\StartingPoint\Module03\LabA.
- Copy the files from that folder to your C:\Scripts folder.

Exercise 1

1. Log on to the Echo VM as Speck with a password of Pa$$w0rd.
2. Open PowerShell ISE as Administrator.
3. Open the C:\Scripts\MyTools.ps1 file.
4. To turn MyTools.ps1 into a module named MyTools, You must:
    - Create a folder for the MyTools module, ensuring that the folder is in the correct location. (C:\Program Files\WindowsPowerShell\Modules\MyTools)
    - Move MyTools.ps1 into the MyTools module folder, renaming the script to MyTools.psm1.
5. Create a MyTools.psd1 module manifest. This must live in the MyTools module folder, and must refer to MyTools.psm1 as the root module.

Spoiler Alert:
Use the New-ModuleManifest command to create a *.psd1 file.

Following is an example of creating and editing a Module Manifest:

```
Cd “C:\Program Files\WindowsPowerShell\Modules\MyTools”

New-ModuleManifest -Path MyTools.psd1 -RootModule MyTools.psm1

ISE MyTools.psm1
```

6. Be sure to check the PSModulePath environment variable to verify that you are using the correct path.

Hint:
To check the PSModulePath environment variable, you can use the following command:

```
Get-Content ENV:\PSModulePath
```

7. When you have finished creating the module, open a new PowerShell console and test the module by ensuring that the following works:

```
Import-Module MyTools

Get-OSInfo –computername ECHO
```

8. If the Get-OSInfo command runs without error, then you completed this lab successfully.
9. You will be modifying MyTools.psm1 in upcoming modules. You can leave the file open in the ISE, but you may have trouble testing it there if you have the functions in memory in PowerShell ISE.
10. After making changes to MyTools.psm1 in the future, you must save the changes and then do one of these:

11. Open a **new** console window, import the module (or let it autoload), and test your commands.
12. Run **Remove-Module MyTools**, then import the module and test your commands.

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module03\LabA

Module 4

Lab A: Implementing Error Handling

(60 minutes)

In this lab, you will work with the MyTools.psm1 module from the last lab in the previous module.
If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab:

- In File Explorer, open C:\LabFiles\StartingPoint\Module04\LabA.
- You will need to copy *all* of the files in that folder to C:\Program Files\WindowsPowerShell\Modules\MyTools.

Exercise 1

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Open PowerShell ISE as Administrator.
3. Open the C:\Program Files\WindowsPowerShell\Modules\MyTools\MyTools.psm1 file.
4. Modify your Get-OSInfo function as follows:
    a. Allow multiple computer names to be provided at the command line.

Hint:
Make sure your $ComputerName parameter accepts multiple values by allowing the variable to be an array of items as follows:

```
[string[]]$Computername
```

    b. Add a ForEach loop to your function, so that you are querying only one computer at a time.

Hint:
Wrap your entire command from the bottom of the Param block to the end of your function in the foreach loop.

```
foreach ($Computer in $ComputerName) { … }
```

Make sure your commands use the $Computer item enumeration variable instead of $ComputerName.

c. Add an –ErrorLog string parameter with a default value of C:\Scripts\Errors.txt

Hint:
Adding a default ErrorLog parameter to the param block would look like:

```
[string]$ErrorLog="C:\Scripts\Errors.txt"
```

d. Add a –LogErrors switch parameter, to the command.

Hint:
A Switch parameter is an on/off value that will equal $True if it is used, or $False if it is not used. The switch parameter would look like the following:

```
[switch]$LogErrors
```

You would use the parameter as follows:

```
Get-OSInfo -ComputerName Alpha -LogErrors
```

a. Add error handling so that failed computer names are logged to a text file if the command was run with a –LogErrors switch and display a warning message if errors occur, even if –LogErrors was not specified.

Hint:
To add error handling, wrap your command in a Try / Catch construct set the ErrorAction of the command to Stop. To conditionally log errors, use an if construct to check the $LogErrors variable and only output to the log if it is true. Write the warning outside the if block so that it always displays if there is an error.

```
Try {
    Get-WmiObject -EA Stop –Class Win32_OperatingSystem –
ComputerName $computer |
        Select-Object
Version,ServicePackMajorVersion,BuildNumber,OSArchitecture
} Catch {
    if ($logerrors) {
        $computer | Out-File $errorlog -append
    }
    Write-Warning "$computer failed"
}
```

5. To test your changes, open a new console window and run:

```
Get-OSInfo –ComputerName ECHO,BAD,BRAVO –LogErrors
```

6. Ensure that C:\Scripts\Errors.txt is created and contains BAD.
7. Delete C:\Scripts\Errors.txt.

Exercise 2

Note:
The same Hints from Exercise 1 will apply to the steps below.

8. Modify your Get-DiskInfo function as follows:
    a. Add a ForEach loop to your function, so that you are querying only one computer at a time.
    b. Add an –ErrorLog string parameter, and a –LogErrors switch parameter, to the command.
    c. Add a default value of C:\Scripts\Errors.txt for the –ErrorLog parameter.
    d. Add error handling, so that failed computer names are logged to a text file if the command was run with a –LogErrors switch.
    e. Display a warning message if errors occur, even if –LogErrors was not specified.

9. To test your changes, open a new console window and run:

```
Get-DiskInfo –ComputerName ECHO,BAD,BRAVO –LogErrors -DriveType 3
```

10. This assumes you defined Get-DiskInfo with a –DriveType parameter. If you used a different parameter name, substitute it in your test command.
11. Ensure that C:\Scripts\Errors.txt is created and contains BAD.
12. Delete C:\Scripts\Errors.txt.

Exercise 3

Note:
The same Hints from Exercise 1 will apply to the steps below.

13. Modify your Invoke-OSShutdown function as follows:
    a. Add a ForEach loop to your function, so that you are querying only one computer at a time.
    b. Add an –ErrorLog string parameter, and a –LogErrors switch parameter, to the command.

c. Add a default value of C:\Scripts\Errors.txt for the –ErrorLog parameter.
d. Add error handling, so that failed computer names are logged to a text file if the command was run with a –LogErrors switch.
e. Display a warning message if errors occur, even if –LogErrors was not specified.

```
Invoke-OSShutdown –ComputerName BAD,BRAVO –LogErrors –Arg 0
```

14. Ensure that C:\Scripts\Errors.txt is created and contains BAD. Delete C:\Scripts\Errors.txt.

Spoiler Alert!
If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module04\LabA

Module 5

Lab A: Implementing Pipeline Input and Output

(60 minutes)

In this lab, you will modify your existing Get-OSInfo and Get-DiskInfo commands. You should have a script from the previous lab to start with.
If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab:

- In File Explorer, open C:\LabFiles\StartingPoint\Module05\LabA
- Copy all of those files to C:\Program Files\WindowsPowerShell\Modules\MyTools.

Exercise 1

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Open a PowerShell prompt as Administrator.
3. To create a Computers.csv file, type the following commands and then press Enter:

```
"ComputerName","Echo","Bravo","Alpha" |
Out-File C:\Scripts\Computers.csv

Import-Csv C:\Scripts\Computers.csv
```

4. Open PowerShell ISE as Administrator.
5. Open the C:\Program Files\WindowsPowerShell\Modules\MyTools\MyTools.psm1 file.

6. Modify Get-OSInfo so that the –ComputerName parameter accepts pipeline input both ByValue and ByPropertyName.

Hint:

To allow the $ComputerName parameter to accept pipeline input, modify the parameter as follows:

```
[Parameter(ValueFromPipeline=$True,
           ValueFromPipelineByPropertyName=$True)]
[string[]]$Computername
```

You must also wrap your code below the end of the param { … } section in a Process { … } block. This will allow the code to be processed once for each of the items coming in through the pipeline.

7. Modify Get-OSInfo so that it continues to query Win32_OperatingSystem, but also queries Win32_ComputerSystem and Win32_BIOS.

Hint:

To query for each of these WMI objects, try the following commands:

```
$OS = Get-WMIObject Win32_OperatingSystem -ComputerName $Computer
$CS = Get-WMIObject Win32_ComputerSystem -ComputerName $Computer
$BIOS = Get-WMIObject Win32_BIOS -ComputerName $Computer
```

8. Have the function output a single unified object that contains:

- The computer name

From Win32_OperatingSystem:

- Version
- ServicePackMajorVersion
- BuildNumber
- OSArchitecture

From Win32_ComputerSystem:

- Manufacturer
- Model

From Win32_BIOS:

- SerialNumber

Hint:

To create a single object with data from multiple sources try the following:

```
$props = @{'ComputerName'=$computer;
           'OSVersion'=$os.version;
           'SPVersion'=$os.servicepackmajorversion;
           'OSBuild'=$os.buildnumber;
           'OSArchitecture'=$os.osarchitecture;
           'Manufacturer'=$cs.manufacturer;
           'Model'=$cs.model;
           'BIOSSerial'=$bios.serialnumber}
$obj = New-Object -TypeName PSObject -Property $props
Write-Output $obj
```

9. Before testing, open a new PowerShell console window.
10. Test your modified script by running all three of the following commands, ensuring that each produces the appropriate output:

```
Remove-Module MyTools

Get-OSInfo –ComputerName BRAVO,ECHO

Import-CSV C:\Scripts\Computers.csv | Get-OSInfo

'BRAVO','ECHO' | Get-OSInfo
```

Exercise 2

11. Modify Get-DiskInfo so that the –ComputerName parameter accepts pipeline input both ByValue and ByPropertyName.

Hint:
To allow the $ComputerName parameter to accept pipeline input, modify the parameter as follows:

```
[Parameter(ValueFromPipeline=$True,
           ValueFromPipelineByPropertyName=$True)]
[string[]]$Computername
```

You must also wrap your code below the end of the param { … } section in a Process { … } block. This will allow the code to be processed once for each of the items coming in through the pipeline.

12. If you have not already done so, modify Get-DiskInfo so that the –DriveType parameter defaults to 3. If you used a parameter name other than –DriveType, modify the appropriate parameter to have a default of 3.
13. Modify Get-DiskInfo so that it continues to query Win32_LogicalDisk.

Hint:

Put the output of Get-WMIObject into a variable as follows:

```
$disks = Get-WmiObject -EA Stop `
                       -Class Win32_LogicalDisk `
                       -Filter "DriveType=$drivetype" `
                       -Computer $computer |
  Where-Object { $_.FreeSpace / $_.Size * 100 -lt $percentfree }
```

14. Have the function output a single unified object that contains:

- The computer name

From Win32_LogicalDisk:

- DeviceID – display this property as "Drive"
- FreeSpace – display this in gigabytes. For example, the formula **($_.FreeSpace / 1GB –as [int])** will produce this result.
- Size – display this in gigabytes. For example, the formula **("{0:N2}" –f ($_.Size / 1GB))** will produce this result.
- FreePercent – calculate this as a percentage. For example, the formula **($_.FreeSpace / $_.Size * 100 –as [int])** will produce this result. You do not need to display a "%" sign.

Hint:

To create a single object with data from multiple sources try the following:

```
foreach ($disk in $disks) {
    $props = @{'ComputerName'=$computer;
               'Drive'=$disk.deviceid;
               'FreeSpace'=($disk.freespace / 1GB -as [int]);
               'Size'=($disk.size / 1GB -as [int]);
               'FreePercent'=($disk.freespace /
                              $disk.size * 100 -as [int])}
    $obj = New-Object -TypeName PSObject -Property $props
```

```
    Write-Output $obj
}
```

15. Before testing, open a new PowerShell console window.
16. Test your modified script by running all three of the following commands, ensuring that each produces the appropriate output:

```
Get-DiskInfo –ComputerName BRAVO,ECHO

Import-CSV C:\Scripts\Computers.csv | Get-DiskInfo

'BRAVO','ECHO' | Get-DiskInfo
```

Exercise 3

17. Modify Invoke-OSShutdown so that the –ComputerName parameter accepts pipeline input both ByValue and ByPropertyName.

Hint:
To allow the $ComputerName parameter to accept pipeline input, modify the parameter as follows:

```
[Parameter(ValueFromPipeline=$True,
           ValueFromPipelineByPropertyName=$True)]
[string[]]$Computername
```

You must also wrap your code below the end of the param { … } section in a Process { … } block. This will allow the code to be processed once for each of the items coming in through the pipeline.

18. Before testing, open a new PowerShell console window.
19. Test your modified script by running all three of the following commands, ensuring that each produces the appropriate output:

Note:
Make sure that Speck is logged on to the console of Alpha and Bravo before running Invoke-OSShutdown with the -Arg 4 parameter.

```
Invoke-OSShutdown –ComputerName ALPHA,BRAVO –Arg 4

'BRAVO','ALPHA' | Invoke-OSShutdown –Arg 2
```

Spoiler Alert!
If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module05\LabA

Module 6

Lab A: Implementing Hierarchical Command Output

(60 minutes)

You should start with the MyTools.psm1 file that you used in the previous lab.
If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab:

- In File Explorer, open: C:\LabFiles\StartingPoint\Modue06\LabA.
- Copy all of the files you find there into C:\Program Files\WindowsPowerShell\Modules\MyTools

Exercise 1

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Open PowerShell ISE as Administrator.
3. Open the C:\Program Files\WindowsPowerShell\Modules\MyTools\MyTools.psm1 file.
4. Create a new function called Get-ComputerVolumeInfo in your existing MyTools module.
5. This function's output will include some information that your other functions already produce, but this particular function is going to combine them all into a single, hierarchical object.
6. This function should accept one or more computer names on a –ComputerName parameter. (Don't worry about error handling at this time.)

Hint:
Allow your $ComputerName parameter to accept multiple values by configuring the variable to be an array of items as follows:

```
[string[]]$Computername
```

Wrap your entire command from the bottom of the Param block to the end of your function in the foreach loop.

```
foreach ($Computer in $ComputerName) { … }
```

Make sure your commands use the $Computer item enumeration variable instead of $ComputerName.

7. Gather the following information from WMI:
   - Win32_OperatingSystem
   - Win32_LogicalDisk a DriveType of 3
   - Win32_Service
   - Win32_Process

Hint:

To query for these WMI objects, try the following code:

```
$os = Get-WmiObject Win32_OperatingSystem -ComputerName $computer
$disks = Get-WmiObject Win32_LogicalDisk -ComputerName $computer `
                                         -Filter "DriveType=3"
$services = Get-WmiObject Win32_Service -ComputerName $computer
$procs = Get-WmiObject Win32_Process -ComputerName $computer
```

8. The output of this function should be a custom object with the following properties:
   - ComputerName
   - OSVersion (Version from Win32_OperatingSystem)
   - SPVersion (ServicePackMajorVersion from Win32_OperatingSystem)
   - LocalDisks (all instances of Win32_LogicalDisk having a DriveType of 3)
   - Services (all instances of Win32_Service)
   - Processes (all instances of Win32_Process)

Hint:

To create the custom hierarchical object, try the following code:

```
$props = @{'ComputerName'=$computer;
           'OSVersion'=$os.version;
           'SPVersion'=$os.servicepackmajorversion;
           'LocalDisks'=$disks;
           'Services'=$services;
           'Processes'=$procs}
$obj = New-Object -TypeName PSObject -Property $props
Write-Output $obj
```

9. The final output should be one object per computer, with each object having 6 properties. Of those properties, 3 will contain sub-objects.
10. Test your script by opening a new PowerShell console window and running:

```
Get-ComputerVolumeInfo –computername ECHO,BRAVO,ALPHA

Get-ComputerVolumeInfo –computername ECHO |
     Select-Object –ExpandProperty Processes
```

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module06\LabA

Module 7

Lab A: Debugging Scripts

(15 minutes)

Exercise 1

We're sure you'll have plenty of practice debugging your own scripts. But we want to reinforce some of the concepts from this chapter and get you used to following a procedure. Never try to debug a script simply by staring at it, hoping the error will jump out at you. It might, but more than likely it may not be the only one. Follow our guidelines to identify bugs. Fix one thing at a time. If it doesn't resolve the problem, change it back and repeat the process.

The functions listed here are broken and buggy. We've numbered each line for reference purposes; the numbers are not part of the actual function. How would you debug them? Revise them into working solutions. Remember, you'll need to dot source the script each time you make a change. We recommend testing in the regular PowerShell console.

The function in the listing below is supposed to display some properties of running services sorted by the service account.

```
Function Get-ServiceInfo {
[cmdletbinding()]
Param([string]$Computername)

$services=Get-WmiObject -Class Win32_Services `
                        -filter "state='Running" `
                        -computername $computername
```

```
Write-Host "Found ($services.count) on $computername" `
           -Foreground Green

$sevices | sort -Property startname,name |
  Select -property startname,name,startmode,computername
}
```

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Open PowerShell ISE as Administrator.
3. Open the sample script from C:\LabFiles\StartingPoint\Module07\LabA\LabA.ps1
4. Examine the script and clean up any messy formatting, fix and debug any errors.

Spoiler Alert!
You can find a debugged version of this function in:
C:\LabFiles\Answers\Module07\LabA\LabA-Debugged.ps1

Lab B: Debugging Scripts

(45 minutes)

Exercise 1

The function in the next script example is a bit more involved. It's designed to get recent event log entries for a specified log on a specified computer. Events are sorted by the event source and added to a log file. The filename is based on the date, computer name, and event source. At the end, the function displays a directory listing of the logs. You may find debugging a lot easier if you clean up the formatting first.

5. Log on to Echo as Speck with a password of Pa$$w0rd.
6. Open PowerShell ISE as Administrator.
7. Open the sample script from C:\LabFiles\StartingPoint\Module07\LabB\LabB.ps1
8. Examine the script and clean up any messy formatting, fix and debug any errors.

Spoiler Alert!

You can find a debugged version of this function in:
C:\LabFiles\Answers\Module07\LabB\LabB-Debugged.ps1

Module 8

Lab A: Implementing Custom Formatting

(60 minutes)

In this lab, you will continue working with your MyTools module.
If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab:

- In File Explorer, open C:\LabFiles\StartingPoint\Module08\LabA.
- Copy all of the files in that location to C:\Program Files\WindowsPowerShell\Modules\MyTools

Exercise 1

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Open PowerShell ISE as Administrator.
3. Open your C:\Program Files\WindowsPowerShell\Modules\MyTools \MyTools.psm1 file.
4. For this lab, you will create a **single** custom view file named C:\Program Files\WindowsPowerShell\Modules\MyTools\MyTools.format.ps1xml. (Later, you will be adding **two** custom views to that file, one for your Get-OSInfo function and one for your Get-DiskInfo function.)
5. Modify the function to add a custom type name to your output object. We suggest MyTools.OSInfo and MyTools.DiskInfo as custom object names.
6. Create the view for each object.
7. Open a fresh PowerShell console window to test your changes.
8. After creating the view file, modify your MyTools.psd1 manifest file so that the view file loads along with the rest of the module.
9. Open MyTools.psd1 in PowerShell ISE and uncomment the "FormatsToProcess" entry.

10. Place the name of your view file inside the parenthesis in quotes after FormatsToProcess.

Exercise 2

11. In the MyTools.format.ps1xml custom view file, create a custom table view for your Get-DiskInfo function.

Hint:

Following is an example of a one column view in a *.ps1xml file:

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
  <ViewDefinitions>
    <View>
      <Name>MyTools.OSInfo</Name>
        <ViewSelectedBy>
          <TypeName>MyTools.OSInfo</TypeName>
        </ViewSelectedBy>
        <TableControl>
          <TableHeaders>
            <TableColumnHeader>
              <Label>ComputerName</Label>
              <Width>15</Width>
            </TableColumnHeader>
          </TableHeaders>
          <TableRowEntries>
            <TableRowEntry>
              <TableColumnItems>
     <TableColumnItem>
       <PropertyName>ComputerName</PropertyName>
                 </TableColumnItem>
              </TableColumnItems>
            </TableRowEntry>
```

```
          </TableRowEntries>
        </TableControl>
      </View>
   </ViewDefinitions>
</Configuration>
```

12. Your table should include all of the data output by the function – try to size your table columns so that they all fit well within a console window that is 80 columns wide.

Exercise 3

13. In the MyTools.format.ps1xml custom view file, create a custom list view for your Get-OSInfo function.
14. We did not cover list views in this module, but you can find examples in Microsoft's files.
15. Open Windows PowerShell as Administrator.
16. Type the following commands to open one of the example view files.

```
cd $pshome

ise dotnettypes.format.ps1xml
```

17. In the file, look for the <ListControl> tags as a reference for building your own list view.

Spoiler Alert!
If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module08\LabA

Module 9

Lab A: Implementing Command Documentation

(30 minutes)

In this lab, you will add comment-based help to all of the functions in the MyTools module. Be sure to open a fresh console window to test your help. If help isn't working, then you've gotten some of the help syntax wrong – nothing will work unless it's completely correct.
You should start with the MyTools module that you worked on in the previous lab. If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab.

- In File Explorer, open: C:\LabFiles\StartingPoint\Module09\LabA.
- Copy all of the files in that folder to C:\Program Files\WindowsPowerShell\Modules\MyTools

Exercise 1

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Open PowerShell ISE as Administrator.
3. Open your C:\Program Files\WindowsPowerShell\Modules\MyTools \MyTools.psm1 file.
4. Now, create a new document inside PowerShell ISE.
5. On the menu, click Edit, Start Snippets, CMDLet (Advanced Function) - Complete. (You can use this template as a reference while adding comment-based help to your functions.)
6. Add comment-based help to Get-OSInfo.

Hint:
Here is a basic comment-based help block that you can use as a reference (place this above your Param block):

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.PARAMETER Param1
   Description of this Parameter
.PARAMETER Param2
   Description of this Parameter
.NOTES
   General notes
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
#>
```

Exercise 2

7. Use the same technique to add comment-based help to Get-DiskInfo.

Exercise 3

8. Use the same technique to add comment-based help to Invoke-OSShutdown, and to any other functions in the MyTools module.

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module09\LabA

Lab B: Implementing Advanced Parameter Attributes

(45 minutes)

In this lab, you will add several advanced attributes to your functions' parameters.
Start with the MyTools.psm1 from the previous lab in this module.
If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab.

- In File Explorer, open: C:\LabFiles\StartingPoint\Module09\LabB
- Copy all of the files in that folder to C:\Program Files\WindowsPowerShell\Modules\MyTools

Exercise 1

9. Log on to Echo as Speck with a password of Pa$$w0rd.
10. Open PowerShell ISE as Administrator.
11. Open your C:\Program Files\WindowsPowerShell\Modules\MyTools \MyTools.psm1 file.
12. Now, create a new document inside PowerShell ISE.
13. On the menu, click Edit, Start Snippets, CMDLet (Advanced Function) - Complete. (You can use this template as a reference while adding advanced parameters to your functions.)
14. Navigate to the Get-OSInfo function in the MyTools.psm1 module.
15. Create an alias –hostname for the –computername parameter
16. Ensure that no more than 5 computer names are provided
17. Make the –computername parameter mandatory
18. Add a help message to all three parameters.
19. Be sure to test your changes.

Exercise 1 – Spoiler Alert!

The following code is what your new Param block could look like:

```
[CmdletBinding()]
param (
    [Parameter(ValueFromPipeline=$True,
```

```
                ValueFromPipelineByPropertyName=$True,
                Mandatory=$True,
                HelpMessage='Computer name or IP address')]
    [Alias('hostname')]
    [ValidateCount(1,5)]
    [string[]]$computername,
    [Parameter(HelpMessage='Default is C:\errors.txt')]
    [string]$errorlog = 'c:\Scripts\errors.txt',
    [Parameter(HelpMessage='Enable failed computer logging')]
    [switch]$logerrors
)
```

Exercise 2

20. Navigate to the Get-DiskInfo function in the MyTools.psm1 module.
21. Create an alias –hostname for the –computername parameter
22. Ensure that no more than 5 computer names are provided
23. Make the –computername and –drivetype parameters mandatory
24. Add a help message to all parameters.
25. Be sure to test your changes.

Exercise 2 - Spoiler Alert!

The following code is what your new Param block could look like:

```
[CmdletBinding()]
param (
    [Parameter(ValueFromPipeline=$True,
                ValueFromPipelineByPropertyName=$True,
                Mandatory=$True,
                HelpMessage='Computer name or IP address')]
    [Alias('hostname')]
    [ValidateCount(1,5)]
    [string[]]$computername,
```

```
    [Parameter(Mandatory=$True,
               HelpMessage='Numeric hard drive type')]
    [int]$drivetype,
    [Parameter(HelpMessage='Percent free space threshold')]
    [int]$percentfree = 99,
    [Parameter(HelpMessage='Default is C:\Errors.txt')]
    [string]$errorlog = 'c:\Scripts\errors.txt',
    [Parameter(HelpMessage='Enable failed computer logging')]
    [switch]$logerrors
)
```

Exercise 3

26. Navigate to the Invoke-OSShutdown function in the MyTools.psm1 module.
27. Rename the –Arg parameter to –Action. Make it accept only the values LogOff, Restart, Shutdown, and PowerOff.
28. Modify the function to take the appropriate action, meaning you will have to translate the value into the appropriate number:
    - For LogOff, use 0
    - For Shutdown, use 1
    - For Restart, use 2
    - For PowerOff, use 8
29. Add a –Force switch parameter. If specified, this should cause your function to add 4 to whatever value it derives form the –Action parameter.
30. Make –Action and –ComputerName mandatory.
31. Add help message to all parameters.

Exercise 3 - Spoiler Alert!

The following code is what your new Param block could look like:

```
[CmdletBinding()]
Param (
```

```
    [Parameter(ValueFromPipeline=$True,
               ValueFromPipelineByPropertyName=$True,
               Mandatory=$True,
               HelpMessage='Computer name or IP address')]
    [string[]]$computername,
    [Parameter(Mandatory=$True,
               HelpMessage='Action to take')]
    [ValidateSet('LogOff','Shutdown','Restart','PowerOff')]
    [string]$action,
    [Parameter(HelpMessage='Force the action')]
    [switch]$force,
    [Parameter(HelpMessage='Default is C:\Errors.txt')]
    [string]$errorlog = 'c:\Scripts\errors.txt',
    [Parameter(HelpMessage='Enable failed computer logging')]
    [switch]$logerrors
)
```

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module09\LabB

Lab C: Implementing Support for ShouldProcess

(15 minutes)

In this lab, you will modify Invoke-OSShutdown only. Start with the MyTools.psm1 module from the previous lab.

If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab:

- In File Explorer, open C:\LabFiles\StartingPoint\Module09\LabC.
- Copy all of the files there to C:\Program Files\WindowsPowerShell\Modules\MyTools

Exercise 1

32. Log on to Echo as Speck with a password of Pa$$w0rd.
33. Open PowerShell ISE as Administrator.
34. Open your C:\Program Files\WindowsPowerShell\Modules\MyTools \MyTools.psm1 file.
35. Modify Invoke-OSShutdown to support ShouldProcess, with a confirm impact of Medium.

Hint:
Following is an example of the changes you need to make in the CmdLetBinding section:

```
[CmdletBinding(SupportsShouldProcess=$True,
                ConfirmImpact='High')]
```

36. Run the following tests in a new console (answer **N**o to all confirmation prompts):

```
$ConfirmPreference = 'Low'

Invoke-OSShutdown –Action LogOff –ComputerName BRAVO
```

37. This test should confirm without needing the –Confirm parameter. Then, try this:

```
$ConfirmPreference = 'High'

Invoke-OSShutdown –Action LogOff –ComputerName BRAVO
```

38. This test should not confirm at all. Finally, try this:

```
$ConfirmPreference = 'Medium'

Invoke-OSShutdown –Action LogOff –ComputerName BRAVO -Confirm
```

39. This should confirm.

Note:
The Invoke-WMIMethod command inside your function will inherit the -WhatIf and -Confirm status of your function. So, you do not need to build ShouldProcess conditional processing into your code body. However, what if you were calling a methad directly? If you wanted to add the ShouldProcess capability, the code would look similar to the following:

```
If ($PSCmdlet.ShouldProcess("Performing $Action on $Computer"))
{
Get-WmiObject -EA Stop –Class Win32_OperatingSystem `
                       –ComputerName $computer |
   Foreach-Object {$_.Win32Shutdown($real_action) | Out-Null}
}
```

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module09\LabC

Module 10

Lab A: Implementing Controller Scripts

(30 minutes)

You should have a MyTools module that contains at least four functions. You created those functions in previous labs.
If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab:

- In File Explorer, open C:\LabFiles\StartingPoint\Module10\LabA.
- Copy all of the files there to C:\Program Files\WindowsPowerShell\Modules\MyTools.

Exercise 1

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Create and save a new file called C:\Scripts\Controller.ps1
3. Declare a dependency on your MyTools module.

Hint:
To declare that a script has a dependency on a specific module, use the following comment directive at the top of the script. (Comments are usually ignored, but this will be processed as a special directive.):

```
#requires -module MyTools
```

4. Design a console-based text menu that allows a technician to run your tools. Your menu should provide the following options:
   - Get system information
   - Get disk information

- Restart a computer

5. The menu script should exhibit the following behavior:
    - The user should enter a number or letter to select one of those three options, which will run your Get-OSInfo, Get-DiskInfo, or Invoke-OSShutdown commands respectively.
    - The user should be prompted for one or more computer names, but should not be prompted for any other information.
    - After running the selected command, your menu should re-display.
    - You can use Ctrl+C to exit the menu.

Spoiler Alert!

Following is an example of the menu code:

```
#requires -module MyTools
while ($true) {
    Write-Host "      SERVICE MENU      "
    Write-Host "========================"
    Write-Host " 1. System Information  "
    Write-Host " 2. Disk Information    "
    Write-Host " 3. Restart a computer  "
    Write-Host ""
    Write-Host "Ctrl+C to exit"
    Write-Host ""
    $choice = Read-Host "Selection"

    if ($choice -eq 1 -or $choice -eq 2 -or $choice -eq 3) {
        $computers = @()
        do {
            $x = Read-Host "Enter one computer name to target;
Enter on a blank line to proceed"
            if ($x -ne '') { $computers += $x }
        } until ($x -eq '')
    }
```

```
    switch ($choice) {
        1 {Get-OSInfo -computername $computers}
        2 {Get-DiskInfo -computername $computers}
        3 {Invoke-OSShutdown -computername $computers -action
Restart}
    }
}
```

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module10\LabA

Module 11

Lab A: Implementing HTML Reports

(45 minutes)

You should have a MyTools module that contains at least four functions. You created those functions in previous labs.
If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab:

- In File Explorer, open C:\LabFiles\StartingPoint\Module11\LabA.
- Copy all of the files there to C:\Program Files\WindowsPowerShell\Modules\MyTools.

Exercise 1

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Open PowerShell ISE as Administrator.
3. Open and save a new file called SystemInformation.ps1
4. In the new file, define a dependency on your MyTools module.

Hint:
To define a dependency, add the following to the top of your script:

```
#requires -module
```

5. Add parameters to accept multiple computer names and a path to output the reports to.

Hint:
Following is an example of the Param block:

```
[CmdletBinding()]
Param(
    [string[]]$ComputerName,
    [string]$path
)
```

6. Add ForEach processing for the multiple computer names.

Hint:
Following is an example of the foreach loop:

```
foreach ($Computer in $ComputerName) {
     # My per-computer report code
}
```

7. Establish an output file name based upon the computer name.

Hint:
The Join-Path command can be used to join a folder and file together into a single path. Following is an example of using Join-Path in your script:

```
$filepath = Join-Path -Path $path -ChildPath "$computer.html"
```

8. Each report section must include its own header.

Hint:
To include a header for each section, use the -PreContent parameter of ConvertTo-HTML and include an <h2> … </h2> html tag around your header text. See the following for an example:

```
ConvertTo-HTML -PreContent '<h2>System Info</h2>'
```

9. Use a variable to retrieve basic System Information, using the output of your Get-OSInfo function pipelined through the ConvertTo-HTML cmdlet as a Fragment.

Hint:
Following is an example of using a variable to store the output of the command above:

```
$sys = Get-OSInfo -computername $computer |
       ConvertTo-HTML -PreContent '<h2>System Info</h2>' `
                      -Fragment | Out-String
```

10. Use a variable to retrieve basic Disk Information, using the output of your Get-DiskInfo function pipelined through the ConvertTo-HTML cmdlet as a Fragment.

Hint:
Following is an example of using a variable to store the output of the command above:

```
$dsk = Get-DiskInfo -computername $computer |
       ConvertTo-HTML -PreContent '<h2>Disk Info</h2>' `
                      -Fragment | Out-String
```

11. Use a variable to store a list of running Processes, using the output of Get-Process pipelined through the ConvertTo-HTML cmdlet as a Fragment. Just include the columns normally displayed when you run Get-Process by itself.

Hint:
Following is an example of using a variable to store the output of the command above:

```
$prc = Get-Process -computername $computer |
    ConvertTo-HTML -PreContent '<h2>Processes</h2>' `
                   -Fragment `
                   -Property Handles,NPM,PM,WS,CPU,ID,SI,Name |
    Out-String
```

12. Use a variable to store a list of installed Services, using the output of Get-Service pipelined through the ConvertTo-HTML cmdlet as a Fragment. Just include the columns normally displayed when you run Get-Service by itself.

Hint:
Following is an example of using a variable to store the output of the command above:

```
$svc = Get-Service -computername $computer |
       ConvertTo-HTML -PreContent '<h2>Services</h2>' `
                      -Fragment `
                      -Property Status,Name,DisplayName |
       Out-String
```

13. Create a CSS style section that makes the formatting more attractive.

Hint:
To include CSS styling to your HTML output, create a variable to store the CSS code and add it to the Head section of the HTML file. See the following for a sample CSS "Here-String".

```
$css = @"
<style>
body {font-family:Tahoma; font-size:12px;}
th {font-weight:bold;}
</style>
"@
```

Note:
The above CSS code is a bit bland, but it reflects what is in the course material and the Answers in the LabFiles. Below is a far more interesting CSS code sample if you would like to use it instead:

```
$css = @"
<style>
body {
    width: 90%;
    margin-top: 10px;
    margin-bottom: 50px;
    margin-left: auto;
    margin-right: auto;
    padding: 0px;
      border-width: 0px;
}
table {
      border-width: 1px;
      border-style: solid;
      border-color: black;
```

```
      border-collapse: collapse;
}
th {
      border-width: 1px;
      padding: 3px;
      border-style: solid;
      border-color: black;
      background-color: lightblue;
}
td {
      border-width: 1px;
      padding: 3px;
      border-style: solid;
      border-color: black;
}
tr:Nth-Child(Even) {
      Background-Color: lightgrey;
}
tr:Hover TD {
      Background-Color: cyan;
}
</style>
"@
```

14. Add the code to assemble all of the data into a single HTML file per computer.

Hint:

Following is an example of the command to generate the HTML report file:

```
$params = @{'Head'="<title>Report for $computer</title>$css";
```

```
            'PreContent'="<h1>Information for $computer</h1>";
            'PostContent'=$sys,$dsk,$prc,$svc}
ConvertTo-Html @params | Out-File -FilePath $filepath
```

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module11\LabA

Module 12

Lab A: Creating a Basic Workflow

(45 minutes)

In this lab there are no dependencies on previous labs. You will be creating a brand new script that will contain a PowerShell Workflow.

Exercise 1

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Open PowerShell ISE as Administrator.
3. Create and save a new script named C:\Scripts\Workflow.ps1.
4. Create a workflow named **Provision1**.

Hint:

Following is an example of the workflow structure:

```
Workflow Provision1 {
    parallel {
        # My Parallel capable code goes here
    }
    Sequence {
        # My code that must run in sequence goes here
        # Hint: Can you create a registry Value before
        #       before the Key exists?
    }
}
```

5. In the workflow, create a registry key named HKEY_LOCAL_MACHINE\SOFTWARE\Custom, and add a string value named Test that has a value of "0".

Hint:
Following is an example of the commands to create the registry key:

```
New-Item -Path HKLM:\SOFTWARE\Custom -Force
New-ItemProperty -Path HKLM:\SOFTWARE\Custom `
                 -Name Test `
                 -Value 0 -Force
```

6. In the workflow, return a list of running services.

Hint:
Following is an example of the command to return a list of running services:

```
Get-Service | Where-Object Status -eq Running
```

7. These tasks can be completed in parallel.
8. Keep in mind that these commands will execute locally on whatever computers the workflow targets, so the commands you write do not need to specify a computer name or credential.
9. You may need to run **Import-Module PSWorkflow** in the ISE's command pane before you begin working on your workflow. Without that module loaded, the workflow components will not work. (In PowerShell 3.0 and above, this should not be necessary.)
10. Save your script file, then click the Play icon or press F5 to run the script.
11. In the blue PowerShell window at the bottom of PowerShell ISE, type the following:

```
Provision1 -PSComputerName ALPHA,ECHO
```

12. The workflow should run on both machines.

Spoiler Alert!
If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module12\LabA

Module 13

Lab A: Working with XML Data

(45 minutes)

You should have a MyTools module that contains at least four functions. You created those functions in previous labs.

If you weren't able to complete the previous lab, do the following to use the Starting Point files for this lab:

- In File Explorer, open C:\LabFiles\StartingPoint\Module131\LabA.
- Copy all of the files there to C:\Program Files\WindowsPowerShell\Modules\MyTools

Exercise 1

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Open PowerShell ISE as Administrator.
3. Create a new file named C:\Scripts\TestData.xml.
4. Type the following code into the script window: (ISE will correctly color code the XML as you type it.)

```
<machines>
  <machine name='Echo' />
  <machine name='Alpha' />
  <machine name='Bravo' />
</machines>
```

5. Open C:\Program Files\WindowsPowerShell\Modules\MyTools\MyTools.psm1 in PowerShell ISE.

6. In your MyTools.psm1 module, add a new function called Get-ComputerNamesFromXML
7. The new function should do the following:
   - Accept the filename of your XML file
   - Output objects that have a ComputerName property containing the values of the Name attributes from the <machine> nodes.

Spoiler Alert!

Following is an example of the Get-ComputerNamesFromXML command:

```
function Get-ComputerNamesFromXML {
  [CmdletBinding()]
  Param(
         [string]$Filename
       )
  [xml]$xml = Get-Content $filename
  foreach ($computer in $xml.machines.machine) {
    $prop = @{'ComputerName'=$computer.name}
    New-Object -TypeName PSObject -Property $prop
  }
}
```

8. Save your changes.
9. In PowerShell ISE, Open the file C:\LabFiles\Answers\Module13\LabA\MyTools\MyTools.psm1
10. Find the function called Set-XMLFileData, highlight and copy the entire function.
11. Paste this function at the end of YOUR MyTools.psm1 file.
12. Save your changes.
13. Examine the new function and ensure that it does the following:
    - Accepts the output of your Get-OSInfo function from the pipeline.
    - Accepts the filename of your XML file.

- Reads the XML file, update it, and save the XML file to the same filename.
- Adds attributes to the <machine> node for every property output by Get-OSInfo, and populate those attributes.

14. Make any changes that you feel may be necessary.
15. To test, open a PowerShell prompt as Administrator and run the following commands:

```
# If you have previously loaded the module, run this command:

Remove-Module MyTools

Get-Content C:\Scripts\TestData.xml

# The next 4 lines are all one command
Get-ComputerNamesFromXML -FileName C:\Scripts\TestData.xml |
   Get-OSInfo | Set-XMLFileData -InputXMLFile
C:\Scripts\TestData.xml -OutputXMLFile
C:\Scripts\NewTestData.xml

Get-Content C:\Scripts\NewTestData.xml
```

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module13\LabA

Module 14

Lab A: Creating a Proxy Function

(45 minutes)

In this lab there are no dependencies on previous labs. You will be creating a brand new script that will contain a proxy function.

Exercise 1

1. Log on to Echo as Speck with a password of Pa$$w0rd.
2. Open PowerShell ISE as Administrator.
3. In the blue PowerShell window at the bottom, type the following commands:

```
$metadata = New-Object
System.Management.Automation.CommandMetaData (Get-Command
Export-CSV)

# The next two lines are all one command
[System.Management.Automation.ProxyCommand]::Create($metadata) |
Out-File C:\Scripts\MyProxy.ps1


ISE C:\Scripts\MyProxy.ps1
```

4. Create a function wrapper around the current script.
5. Name the function Export-TDF

Hint:
Following is an example of the function declaration:

```
function Export-TDF {
    # Here is the existing code from the original command
}
```

6. Remove the –Delimiter parameter.

Hint:
Following is the parameter that needs to be removed:

```
[Parameter(ParameterSetName='Delimiter', Position=1)]
[ValidateNotNull()]
[char]
${Delimiter},
```

7. Now hardcode the function to always use a Tab delimited format. (–Delimiter "`t" - that's a backtick, followed by the letter *t*, in double quotation marks).
8. Save the file.
9. Click the Play icon or press F5 to run the script.
10. In the blue PowerShell window at the bottom, type the following:

```
Get-Service | Export-TDF c:\Scripts\services.tdf

Notepad C:\Scripts\services.tdf
```

11. The resulting file should contain Tab-delimited data.

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module14

Module 15

Lab A: PowerShell Toolmaking

(120 minutes)

This is the lab scenario. There are no lab steps here, try to do this on your own. If you want more detail, check out the section that follows this one. You will find steps that will help you along the way.

For this lab, you will not be reusing any of your prior work. You will be creating an entirely new module named CorpTools.

The CorpTools module should consist of three tools:

a. Get-NetAdaptInfo This should query Win32_NetworkAdapter from WMI, including only physical adapters (PhysicalAdapter=True). Display the AdapterType, Speed (in gigabytes, but do not include a 'GB' identifier), the NetConnectionID, and the MACAddress. Keep in mind that a computer may have more than one adapter installed. The computer name should also be included as a property of each adapter.
b. Get-SystemInfo This should query the following classes and return a unified object containing:
    - From Win32_ComputerSystem, include the computer name, DNSHostName, AutomaticManagedPagefile, Manufacturer, Model, and Domain.
    - From Win32_BIOS, include the SerialNumber, naming the property BIOSSerial.
    - From Win32_BaseBoard, include the Product (naming the property BaseBoardProduct) and Manufacturer (naming the property BaseBoardMfgr).
c. Get-StartedServices This should query all running services, using either Get-Service, or the Win32_Service WMI class. Include the

computer name, the service name, and the service's executable filename.

Test each of your tools, create a custom format view for the module, ensuring that it will load as part of the module manifest. The view file needs to include a table view for the output of Get-NetAdaptInfo, and a list view for Get-SystemInfo.

Those views must include:

- For Get-NetAdaptInfo, AdapterType, Speed, NetConnectionID, and MACAddress. Size the columns to minimize wasted screen space.
- For Get-SystemInfo, include only DNSHostName, Manufacturer, Model, and BaseBoardMfgr.

After completing and testing the tools in the module, create a controller script named SystemReport2.ps1. This script should accept one (and only one) computer name on a –ComputerName parameter.

It should produce a nicely-formatted HTML report consisting of three sections:

- Network Adapters (the full output of Get-NetAdaptInfo)
- System Information (the full output of Get-SystemInfo, in a table)
- Running Services (the full output of Get-RunningServices, in a table)

A second parameter, -OutFilePath, should accept the path and filename for the HTML report.

If you would like the more detailed approach to the lab, below are the lab steps that will do that for you. Read the scenario above thoroughly so that you understand what you need to accomplish, then refer to the exercises below for the step-by-step process.

Exercise 1: Create the Module and Functions

Task 1: Create the CorpTools module

1. Log on to Echo as Speck with a password of Pa$$word.
2. Open PowerShell as Administrator.
3. Create a module folder in C:\Program Files\WindowsPowerShell\Modules called CorpTools.
4. Open PowerShell ISE as Administrator.

```
md “C:\Program Files\WindowsPowerShell\Modules\CorpTools”

# The command below will launch PowerShell ISE
# in the same security context as this PowerShell
# prompt.

ise
```

5. Create and open a new file called CorpTools.psm1 in the C:\Program Files\WindowsPowerShell\Modules\CorpTools folder.

Task 2: Create the Get-NetAdaptInfo function

6. In the CorpTools module, create a function called Get-NetAdaptInfo.
7. Add a param block for a $ComputerName parameter.
8. Add a command to query the Win32_NetworkAdapter object for Physical Adapters only from the remote computer and put the result in a variable.
9. Use a foreach construct to loop through the array variable you just created, and output a custom object with the following properties:
    - ComputerName
    - Speed
    - AdapterType
    - NetConnectionID

- MACAddress

10. Assign a unique object type to your new object.
11. Write the object to the pipeline.

Spoiler Alert!

Following is an example of the Get-NetAdaptInfo function:

```
function Get-NetAdaptInfo {
    [CmdletBinding()]
    Param(
        [string]$ComputerName
    )
    $adapts = Get-WmiObject -Class Win32_NetworkAdapter `
                            -Filter "PhysicalAdapter='True'" `
                            -ComputerName $computername
    foreach ($adapt in $adapts) {
        $props = @{'ComputerName'=$computername;
                   'Speed'=($adapt.speed / 1GB -as [int]);
                   'AdapterType'=$adapt.adaptertype;
                   'NetConnectionID'=$adapt.netconnectionid;
                   'MACAddress'=$adapt.macaddress}
        $obj = New-Object -TypeName PSObject -Property $props
        $obj.psobject.typenames.insert(0,'CorpTools.NetAdaptInfo')
        Write-Output $obj
    }
}
```

Task 3: Create the Get-SystemInfo function

12. In the CorpTools module, create a function called Get-SystemInfo.
13. Add a param block for a $ComputerName parameter.

14. Add a command to query the Win32_ComputerSystem object and put the result in a variable.
15. Add a command to query the Win32_BIOS object and put the result in a variable.
16. Add a command to query the Win32_Baseboard object and put the result in a variable.
17. Create a custom object with the following properties:
    - ComputerName
    - DNSHostName
    - AutomaticManagedPageFile
    - Manufacturer
    - Model
    - Domain
    - BIOSSerial
    - BaseBoardProduct
    - BaseBoardMfgr

18. Assign a unique object type to your new object.
19. Write the object to the pipeline.

Spoiler Alert!

Following is an example of the Get-SystemInfo function:

```
function Get-SystemInfo {
    [CmdletBinding()]
    Param(
        [string]$ComputerName
    )
    $cs = Get-WmiObject -Class Win32_ComputerSystem `
                        -ComputerName $computername
    $bios = Get-WmiObject -Class Win32_BIOS `
                          -ComputerName $computername
```

```
    $bb = Get-WmiObject -Class Win32_BaseBoard `
                        -ComputerName $computername
    $props = @{'ComputerName'=$computername;
               'DNSHostName'=$cs.dnshostname;
'AutomaticManagedPageFile'=$cs.automaticmanagedpagefile;
               'Manufacturer'=$cs.manufacturer;
               'Model'=$cs.model;
               'Domain'=$cs.domain;
               'BIOSSerial'=$bios.serialnumber;
               'BaseBoardProduct'=$bb.product;
               'BaseBoardMfgr'=$bb.manufacturer}
    $obj = New-Object -TypeName PSObject -Property $props
    $obj.psobject.typenames.insert(0,'CorpTools.SystemInfo')
    Write-Output $obj
}
```

Task 4: Create the Get-StartedServices function

20. In the CorpTools module, create a function called Get-StartedServices.
21. Add a param block for a $ComputerName parameter.
22. Add a command to query the Win32_Service object for running services only from the remote computer and put the result in a variable.
23. Use a foreach construct to loop through the array variable you just created, and output a custom object with the following properties:
    - ComputerName
    - ServiceName
    - Executable

24. Assign a unique object type to your new object.
25. Write the object to the pipeline.

Spoiler Alert!

Following is an example of the Get-StartedServices function:

```
function Get-StartedServices {
    [CmdletBinding()]
    Param(
        [string]$ComputerName
    )
    $services = Get-WMIObject -Class Win32_Service `
                              -Filter "State='Running'" `
                              -computername $ComputerName
    foreach ($service in $services) {
        $props = @{'ComputerName'=$ComputerName;
                   'ServiceName'=$service.name;
                   'Executable'=$service.pathname}
        $obj = New-Object -TypeName PSObject -Property $props
        $obj.psobject.typenames.insert(0,'CorpTools.RunningService')
        Write-Output $obj
    }
}
```

Exercise 2: Create the Module Manifest and Custom View

Task 1: Create the Module Manifest

26. In PowerShell ISE, go to the blue PowerShell window and create a Module Manifest for the CorpTools module that references the PSM1 and PS1XML files. (See the command below)

```
cd “\Program Files\WindowsPowerShell\Modules\CorpTools”
```

```
# The command below is a single command
New-ModuleManifest -Path CorpTools.psd1
-RootModule CorpTools.psm1 -FormatsToProcess
CorpTools.format.ps1xml
```

27. Open the new Module Manifest file in PowerShell ISE and make sure that all of the settings are correct and edit any options as you see fit.

Task 12 Create the Custom View

28. In PowerShell ISE, create a new document and save it as: C:\Program Files\WindowsPowerShell\Modules\CorpTools\CorpTools.format.ps1xml
29. Create the basic framework for the format file.

Hint:

Following is the basic XML code to establish a format file

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
    <ViewDefinitions>
     # Custom View code goes here
    </ViewDefinitions>
</Configuration>
```

30. Create a custom table view for the NetAdaptInfo object that includes AdapterType, Speed, NetConnectionID, and MACAddress.

Spoiler Alert!

Following is the XML code for the NetAdaptInfo table view:

```
<View>
```

```
    <Name>CorpTools.NetAdaptInfo</Name>
    <ViewSelectedBy>
        <TypeName>CorpTools.NetAdaptInfo</TypeName>
    </ViewSelectedBy>
    <TableControl>
        <TableHeaders>
            <TableColumnHeader>
                <Label>AdapterType</Label>
            </TableColumnHeader>
            <TableColumnHeader>
                <Label>Speed(Gbps)</Label>
            </TableColumnHeader>
            <TableColumnHeader>
                <Label>NetConnectionID</Label>
            </TableColumnHeader>
            <TableColumnHeader>
                <Label>MACAddress</Label>
            </TableColumnHeader>
        </TableHeaders>
        <TableRowEntries>
            <TableRowEntry>
                <TableColumnItems>
                    <TableColumnItem>
                        <PropertyName>AdapterType</PropertyName>
                    </TableColumnItem>
                    <TableColumnItem>
                        <PropertyName>Speed</PropertyName>
                    </TableColumnItem>
                    <TableColumnItem>
                    <PropertyName>NetConnectionID</PropertyName>
                    </TableColumnItem>
                    <TableColumnItem>
                        <Propertyname>MACAddress</Propertyname>
```

```
                    </TableColumnItem>
                </TableColumnItems>
            </TableRowEntry>
            </TableRowEntries>
    </TableControl>
</View>
```

31. Create a custom list view for the SystemInfo object that includes DNSHostName, Manufacturer, Model and BaseBoardMfgr

Spoiler Alert!

Following is the XML code to create a list view for the SystemInfo object:

```
<View>
    <Name>CorpTools.SystemInfo</Name>
    <ViewSelectedBy>
        <TypeName>CorpTools.SystemInfo</TypeName>
    </ViewSelectedBy>
    <ListControl>
        <ListEntries>
            <ListEntry>
                <ListItems>
                    <ListItem>
                        <PropertyName>DNSHostName</PropertyName>
                    </ListItem>
                    <ListItem>
                       <PropertyName>Manufacturer</PropertyName>
                    </ListItem>
                    <ListItem>
                        <PropertyName>Model</PropertyName>
                    </ListItem>
                    <ListItem>
```

```
                    <PropertyName>BaseBoardMfgr</PropertyName>
                </ListItem>
            </ListItems>
        </ListEntry>
    </ListEntries>
</ListControl>
</View>
```

32. Save the file.

Exercise 3: Create a Controller Script to produce an HTML report

Task 1: Create a controller script file with a param block

33. In PowerShell ISE, create a new document and save it as: C:\Scripts\SystemReport2.ps1
34. Create a param block to accept a ComputerName and an OutFilePath parameter.

Spoiler Alert:
Following is an example of the param block:

```
[CmdletBinding()]
Param(
    [string]$ComputerName,
    [string]$OutFilePath
)
```

Task 2: Create the HTML content

35. Create a variable to store the HTML output of Get-NetAdaptInfo as a "Network Adapters" section.

Spoiler Alert!

Following is an example of the Network Adapters section code:

```
$na = Get-NetAdaptInfo -computername $ComputerName |
      ConvertTo-HTML -PreContent '<h2>Network Adapters</h2>' `
                     -Fragment | Out-String
```

36. Create a variable to store the HTML output of Get-SystemInfo as a "System Information" section.

Spoiler Alert!

Following is an example of the SystemInfo section code:

```
$si = Get-SystemInfo -computername $ComputerName |
      ConvertTo-HTML -PreContent '<h2>SystemInfo</h2>' `
                     -Fragment | Out-String
```

37. Create a variable to store the HTML output of Get-RunningServices as a "Running Services" section.

Spoiler Alert!

Following is an example of the Running Services section code:

```
$rs = Get-StartedServices -computername $ComputerName |
      ConvertTo-HTML -PreContent '<h2>Running Services</h2>' `
                     -Fragment | Out-String
```

38. As a bonus, try to include style sheet code within your HTML output in order to make the results more visually appealing. (Note that you will not find this in the answer files.)

Bonus:

To include CSS styling to your HTML output, create a variable to store the CSS code and add it to the Head section of the HTML file. See the following for a sample CSS "Here-String".

```
$css = @"
<style>
body {
    width: 90%;
    margin-top: 10px;
    margin-bottom: 50px;
    margin-left: auto;
    margin-right: auto;
    padding: 0px;
      border-width: 0px;
}
table {
      border-width: 1px;
      border-style: solid;
      border-color: black;
      border-collapse: collapse;
}
th {
      border-width: 1px;
      padding: 3px;
      border-style: solid;
      border-color: black;
      background-color: lightblue;
}
td {
      border-width: 1px;
      padding: 3px;
      border-style: solid;
```

```
      border-color: black;
}
tr:Nth-Child(Even) {
      Background-Color: lightgrey;
}
tr:Hover TD {
      Background-Color: cyan;
}
</style>
"@
```

Task 3: Output the HTML results to a file

39. Assemble the report variables together using the ConvertTo-HTML command and produce a report file.

Spoiler Alert!

Following is an example of the ConvertTo-HTML code:

```
ConvertTo-HTML -PreContent "$computername Information" `
               -PostContent $na,$si,$rs | Out-File $OutFilePath
```

Or, if you went through the Bonus step of creating CSS code, use the following command instead:

```
ConvertTo-HTML -Head $css `
               -PreContent "$computername Information" `
               -PostContent $na,$si,$rs | Out-File $OutFilePath
```

Task 4: Test the HTML reporting script

40. Assemble the report variables together using the ConvertTo-HTML command and produce a report file.
41. Save the SystemReport2.ps1 file
42. Open a PowerShell prompt as Administrator.
43. Type the following commands to test your script:

```
cd C:\Scripts

SystemReport2.ps1 Alpha -OutFilePath C:\Scripts\AlphaReport.htm

SystemReport2.ps1 Bravo -OutFilePath C:\Scripts\BravoReport.htm

.\AlphaReport.htm

.\BravoReport.htm
```

44. View the reports in the browser and see if you need to make adjustments to your module or your controller script.
45. If everything looks good, congratulations! You have finished the project!

Spoiler Alert!

If you would like to compare your results to the answer files, or if you are stuck and need to peek at the solution for some insight, you will find the files at: C:\LabFiles\Answers\Module15

Congratulations!
You have completed all of the lab steps in this course!

If you would like to preserve any of your work, the labs should have access to the internet, and you should be able to copy your files out to a cloud-based drive, or attach you files as a ZIP package to an email in a web-based mail service.

If that doesn't work for you, just bring your code up in PowerShell ISE and capture screenshots of your work.

Thank you for attending the course!