Windows PowerShell 4.0 For the IT Professional - Part 1

Module 9: Pipeline 2

Student Lab Manual

Version 2.0

Conditions and Terms of Use

Microsoft Confidential - For Internal Use Only

Contents

Lab 9: Pipeline 2

Introduction

So far, you have been using the Windows PowerShell pipeline in its simplest form. In this module, we will explore the pipeline syntax and usage in more depth.

Objectives

After completing this lab, you will be able to:

- Use different names to reference the current pipeline object
- Filter and iterate objects with Where-Object & Foreach-Object Cmdlets
- Control the execution of pipeline processing with Foreach-Object parameters
- Understand how Windows PowerShell binds pipelined parameters

Prerequisites

Start all VMs provided for the workshop labs.

Logon to WIN8-WS as:

Username: **Contoso\Administrator**

Password: **PowerShell4**

Estimated time to complete this lab

90 minutes

NOTE: These exercises use many Windows PowerShell commands. You can type these commands into the Windows PowerShell Integrated Scripting Environment (ISE) or the Windows PowerShell console. For some exercises, you can load pre-typed lab files into the Windows PowerShell ISE, allowing you to select and execute individual commands. Each lab has its own folder under **C:\PShell\Labs** on **WIN8-WS**.

Some exercises in this workshop may require running the Windows PowerShell console or ISE as an elevated user (Run as Administrator).

We recommend that you connect to the virtual machines (VMs) for these labs through Remote Desktop rather than connecting through the Hyper-V console. This allows you to use copy and paste between VMs and the host machine. If you are using the online hosted labs, then you are already using Remote Desktop.

Exercise 9.1: Pipeline Objects

Introduction

When performing pipeline operations in previous lab exercises, we observed how easily lists of objects can be sorted, selected, formatted and output to structured file formats. In this exercise, we will learn how to perform more complex filtering on pipeline objects.

Objectives

After completing this exercise, you will be able to:

- Use different variable names to refer to the current pipeline object
- Filter and iterate pipeline objects with Where-Object & ForEach-Object Cmdlets

Scenario

You have been given a character separated (CSV) text file containing the names of 104 new employees at Contoso. You would like to use Windows PowerShell to import the file and create each batch of users. First you must create the organizational units (OUs).

Task 9.1.1: Pipeline variable

1. List three ways you can reference the current pipeline variable?

   a) $_

   b) $PSItem

   c) -PipelineVariable common parameter allows for custom naming of the current pipeline variable

2. Create a comma-separated list of strings. Send them through the pipe operator to the ForEach-Object Cmdlet. This Cmdlet accepts commands within a scriptblock. Each string passed to the scriptblock can be accessed through the current pipeline object variable ($_).

   NOTE: The Windows PowerShell pipeline processes a single object at a time. As shown below, the current object being processed is accessed using the special pipeline variable "$_".

   ```
   "Engineering-AU","Marketing-AU","IT-AU","Finance-AU" | ForEach-Object
   { $_ }
   ```

3. Using the Get-Member Cmdlet, examine the type of objects used as pipeline input.

```
"Engineering-AU","Marketing-AU","IT-AU","Finance-AU" | Get-Member
```

4. In this example, what type of object is stored in the pipeline variable?

System.String

5. Members on the current string in the pipeline can be accessed, using the dot operator. Use the String object's Trim() method to remove the "-AU" characters from each string.

```
"Engineering-AU","Marketing-AU","IT-AU","Finance-AU" |
ForEach-Object { $_.Trim("-AU") }
```

Note that Trim() acts upon a SET of characters!
"Removes all leading and trailing occurrences of a set of characters specified in an array from the current String object."
This can sometimes behave in ways you didn't expect.

6. Find Cmdlets that manage OrganizationalUnit objects.

```
Get-Command -Noun *OrganizationalUnit
```

Get-ADOrganizationalUnit
New-ADOrganizationalUnit
Remove-ADOrganizationalUnit
Set-ADOrganizationalUnit

7. Review the help and examples for the New-ADOrganizationalUnit Cmdlet.
8. Create a new OU, named "Personnel", in the following path "DC=Contoso,DC=com".

```
New-ADOrganizationalUnit -Path "DC=Contoso,DC=com" -Name Personnel
```

9. Within the new OU, create four child OUs, using the strings in step 5.

```
"Engineering-AU", "Marketing-AU","IT-AU", "Finance-AU"
```

NOTE: This could be achieved a number of ways. For example, we could simply type the New-ADOrganizationalUnit Cmdlet four times using each of the OU names as the value for the New-ADOrganizationlUnit Cmdlets's –Name parameter, but the Windows PowerShell pipeline makes this far easier.

Use the ForEach-Object Cmdlet and the current pipeline object variable to create the four new OUs. Use the String object's Trim() method to remove the trailing "-AU" characters from each string during the creation of each OU.

```
"Engineering-AU","Marketing-AU","IT-AU","Finance-AU" |
ForEach-Object {
New-ADOrganizationalUnit -Path "OU=Personnel,DC=Contoso,DC=com" `
-Name $_.Trim("-AU")
}
```

Execute the following command to list the newly created OUs.

```
Get-ADOrganizationalUnit -SearchBase "OU=Personnel,DC=Contoso,DC=com" -
Filter * |
Select-Object -Property Name, DistinguishedName
```

Exercise 9.2: Pipeline Cmdlets

Introduction

As we have already seen, the pipeline variable is used within the ForEach-Object Cmdlet's scriptblock in order to access the current object being processed in the pipeline. The ForEach-Object allows an action to be taken for each of a number of objects. The Where-Object Cmdlet provides a powerful way to filter pipeline objects.

Objectives

After completing this exercise, you will be able to:

- Use the Foreach-Object Cmdlets to iterate pipeline objects.
- Use the Where-Object Cmdlet to filter pipeline objects.

Scenario

It is your responsibility to create new user accounts in the Contoso Active Directory correctly and in a timely manner. You have received a CSV file containing the details of over 100 new employees and now need to quickly, and accurately, create them in Active Directory.

Task 9.2.1: Filtering pipeline objects

1. The Windows PowerShell pipeline can filter using the Where-Object Cmdlet. This Cmdlet depends on the result of one or more operators being either true or false. If the result is true, the object passes along to the next stage of the pipeline. If it is false, the object is discarded.

   In the simplest form this can be achieved using the built-in $true or $false variables. Type the following commands. The first command displays all objects emitted by the Get-Process Cmdlet.

```
Get-Process | Where-Object {$true}
```

   The second command prevents the objects being emitted from the pipeline.

```
Get-Process | Where-Object {$false}
```

2. Type the path to the script C:\PShell\Labs\Lab_9\Generate-UsersXML.ps1, in the Windows PowerShell ISE command pane, then press enter. A new file, C:\PShell\Labs\Lab_9\Contoso-UserImport.xml will be created. We will use the user information in this file to create new Active Directory users.
3. Import the new user details from the xml file in C:\PShell\Labs\Lab_9\Contoso-UserImport.xml and display them in the Windows PowerShell ISE or console.

```
Import-CliXml -Path C:\PShell\Labs\Lab_9\Contoso-UserImport.xml
```

4. Get the member details for the particular object type produced by the Import-CliXml Cmdlet.

```
Import-CliXml -Path C:\PShell\Labs\Lab_9\Contoso-UserImport.xml | Get-Member
```

5. We can filter the list of users, using the Where-Object Cmdlet, the pipeline variable, comparison operators, and logical operators. Type and execute the examples below.

```
Import-CliXml -Path C:\PShell\Labs\Lab_9\Contoso-UserImport.xml -Outvariable
userList
```

6. Use the Where-Object Cmdlet to filter the user objects.

```
$userList | Where-Object {$_.department -eq "IT"}
$userList | Where-Object {$_.LastName -eq "Smith"}
$userList | Where-Object {$_.department -eq "Finance" -and $_.LastName -eq
"Hollinworth"}
```

7. Create pipeline commands to gather the following information. Write the command below each bulleted item.

   a) Users where the "Division" property equals "Widgets".

```
$users | Where-Object {$_.division -eq
"Widgets"}
```

   b) Users where the Division equals "Gadgets" and the Department equals "Engineering".

```
$users | Where-Object {$_.division -eq
"Gadgets" -and $_.department -eq
"Engineering"}
```

   c) Users whose AccountExpirationDate property is less than or equal to 6 months from today's date.

```
$In6Months = (Get-Date).Addmonths(6)

$users | Where-Object
{$_.accountexpirationdate -le $In6Months}
```

TIP: Using Get-Help and Get-Member, explore the Get-Date Cmdlet for a method to add 6 months to the current date.

8. Use the Department property value to determine the OU in which each user should be created. The following command will import the information in the file and then filter users whose "Department" property contains the string "Marketing".

```
Import-CliXml -Path C:\PShell\Labs\Lab_9\Contoso-UserImport.xml |
Where-Object {$_.department -eq "Marketing"}
```

9. We can now create user accounts in Active Directory. The New-ADUser Cmdlet creates a new user for each object passed to it. Notice that we specify the current object's Department and Name parameters as values for the New-ADUser Cmdlet's -Department and -Name parameters.

```
Import-CliXml -Path C:\PShell\Labs\Lab_9\Contoso-UserImport.xml |
Foreach-Object {
    New-ADUser -Path "OU=$($_.Department),OU=Personnel,DC=Contoso,DC=com" `
        -Name $_.Name –GivenName $_.FirstName
}
```

10. Verify 104 users were created successfully.

```
(Get-ADUser -SearchBase "OU=Personnel,DC=Contoso,DC=com" -Filter *).Count
```

Task 9.2.2: ForEach-Object and Where-Object simplified syntax

1. Both the Where-Object and ForEach-Object Cmdlets implement an alternative, simplified, syntax that does not require the script block curly braces or the pipeline variable. Compare the two syntaxes below. Each command has exactly the same behavior.

```
Import-Clixml -Path C:\PShell\Labs\Lab_9\Contoso-UserImport.xml |
Where-Object {$_.department -like "*Marketing*"}

Import-Clixml -Path C:\PShell\Labs\Lab_9\Contoso-UserImport.xml |
Where-Object department -like "*Marketing*"
```

2. Re-write the queries shown below using the alternative, simplified syntax.

```
$userList | Where-Object {$_.department -eq IT}
```

$userList | Where-Object Department -eq IT

```
$userList | Where-Object {$_.LastName -eq Smith}
```

```
$userList | Where-Object {$_.Division –eq Widgets}
```

---

3. Using the simplified syntax, return users where the Division equals "Widgets" and the Department equals "Marketing". Write the command below.

```
$userList | Where-Object {$_.division -eq "Widgets" -and $_.Department -eq
"Marketing"}
```

Were you able to use the simplified syntax to achieve this (circle the response)?

Yes / No

NO. Logical operators, such as -And and -Or, are valid only in script blocks. You cannot use them in the comparison statement format of a Where-Object command.

TIP: The simplified syntax for the Where-Object Cmdlet does not support logical operators (-and, -or, xor, etc.).

4. Using the simplified syntax, return users whose AccountExpirationDate property is less than or equal to 6 months from today's date.

```
$In6Months = (Get-Date).Addmonths(6)

$users | Where-Object accountexpirationdate -
le $In6Months
```

Exercise 9.3: ForEach-Object Pipeline Processing

Introduction

The ForEach-Object Cmdlet has the ability to manipulate pipeline data processing at three stages in the pipeline.

1. **Begin** - Before the Cmdlet has processed any objects
2. **Process** - For each object processed
3. **End** - Once the Cmdlet has processed all objects.

Objectives

After completing this exercise, you will be able to:

- Control the execution of pipeline processing with ForEach-Object parameters

Task 9.3.1: Script block parameters

1. The ForEach-Object Cmdlet implements nested script blocks exposed through its –Begin, -Process and –End parameters. Commands can be executed at different stages of a pipeline operation.
   Type and execute the command below.

```
Get-Service |
Foreach-Object –begin {"Counting Services..." ; $count=0} `
–process {$count++} `
–end {“$count services were found”}
```

2. As with any Cmdlet that accepts a script block data type, the –begin, -process, –end blocks can alternatively accept variables containing script block objects. Type and execute the script below.

```
$process = {
        "Restarting Computer: $_"
        Restart-Computer -ComputerName $_ -Wait -For WinRM -WhatIf
        $count++
}

"2012R2-DC", "2012R2-MS" |
ForEach-Object -begin {$count = 0} -process $process -end {"Restarted $count
computers"}
```

Task 9.3.2: Sending pipeline input to a function

Your own functions also have the ability to accept pipeline input.

1. Open the file C:\PShell\Labs\Lab_9\Get-NetAdapterInfo.ps1 in the Windows PowerShell ISE. The script is shown below.

```
function Get-NetAdapterInfo {

Param($timeout = 10)

begin {$count = 0}

process {

Write-Output "Processing computer: $_"

$session = New-CimSession –ComputerName $_ -OperationTimeoutSec $timeout

Get-NetAdapter –cimsession $session

$count++

}

end {Write-Output "$count computers were processed"}

}
```

Define the function by pressing the 'play' button within the ISE, or by pressing the f5 key. Test the function in the ISE command pane by sending input to it through the pipeline.

```
“2012R2-DC”,”2012R2-MS”,”WIN8-WS” | Get-NetAdapterInfo
```

2. Extend the pipeline command in the previous step to select the following properties PSComputerName, DriverName, InterfaceName and InterfaceIndex, then convert the pipeline output to Html format and save it in C:\PShell\Labs\Lab_9\NetAdapterInfo.htm.

   Write the pipeline command you used to achieve this below.

```
"2012R2-DC","2012R2-MS","WIN8-WS" |
Get-NetAdapterInfo |
Select-Object
PSComputername,DriverName,InterfaceName,Interf
aceIndex |
ConvertTo-Html |
Out-File C:\PShell\Labs\Lab_9\
NetAdapterInfo.htm
```

3. Open the C:\PShell\Labs\Lab_9\NetAdapterInfo.htm file that you just created.

```
Invoke-Item C:\PShell\Labs\Lab_9\NetAdapterInfo.htm
```

Task 9.3.3: Using Script blocks in the pipeline

1. A script block can contain begin, process & end nested script blocks just like the ForEach-Object cmdlet. The child script blocks must be defined within the parent script block body and are executed at different stages in the pipeline.

   The extended syntax for a script block is below.

```
{
param( $parameter1, $parameter2 )

       begin { <statement list> }
       process{ <statement list> }
       end { <statement list> }
}
```

2. In the example, the Get-Service Cmdlet outputs a list of ServiceController objects, then:

   a) The begin block creates a new variable called $count and sets its value to 0.
   b) The process block increments $count by one for each object received.
   c) The end block returns the total number of service objects.

   Open C:\PShell\Labs\Lab_9\9.3.3.2.ps1 in the Windows PowerShell ISE. Press the 'play' button or <F5> to execute it.

   The ampersand (&) character is known as the "invoke operator", and it will be covered later.

```
Get-Service | & {
begin {"Counting Services..." ; $count=0}
process {$count++}
end {"$count services were found"}
}
```

Exercise 9.4: Pipeline Parameter Binding

Introduction

Some Cmdlet parameters allow their parameter values to be populated from the previous Cmdlet in the pipeline.

Objectives

After completing this exercise, you will be able to:

- Understand how Windows PowerShell binds parameters in a pipeline
- Use this knowledge to overcome Cmdlet pipeline limitations

Scenario

You now need to populate more information contained in the Contoso-UserImport.xml file for each user during the user creation process.

We can take advantage of Windows PowerShell's parameter binding behavior to make this much easier.

Task 9.4.1: Parameter Binding

1. There are two ways parameter binding can be achieved:
   - By Value
   - By Property Name

   Get help for the about_parameters conceptual topic,

```
Get-Help About_Parameters –ShowWindow
```

   Scroll down to the "Accepts Pipeline Input?" header and read the descriptions.

2. If a property is marked as "True (by Value)", what order does Windows PowerShell associate the piped values?

   When a parameter is "True (by Value)", Windows PowerShell tries to associate any piped values with that parameter **before** it tries other methods to interpret the command.

3. Find the Stop-Service Cmdlets –Name Parameter help.

```
Get-Help Stop-Service -Parameter Name
```

4. Does this parameter accept pipeline input? Circle the correct answer.

   Yes / No

5. Does the parameter accept values bound by value, by property name or both?

   _____

6. Try the following commands and determine whether they are successful or not. Note that the –Passthru switch parameter allows the service controller objects that were started to be displayed in the console.

```
Get-Service spooler | Start-Service –PassThru
"spooler" | Stop-Service -PassThru
"spooler" | Get-service | Select-Object name | Stop-Service -PassThru
"spooler" | Start-Service -PassThru
```

7. Look at the second example in more detail.

```
“spooler” | Stop-Service -Passthru
```

   Get help for the Stop-Service Cmdlet’s Name parameter.

```
Help Stop-Service –Parameter Name
```

   Does the parameter accept pipeline input?

   Yes / No

8. The ByValue reference means that the Start-Service Cmdlet will accept pipeline input of a String type and automatically bind it as the value of the –Name parameter.

   NOTE: A Cmdlet will only have one parameter of a given data type that will accept pipeline input ByValue. This removes ambiguity during parameter binding.

   How would you start both the “Spooler” and “WSearch” services using this method? Write the solution below.

   NOTE: Get help for the Start-Service’s –Name parameter and determine whether it accepts a single or multiple values.

   _____

   _____

YES

Both byValue & byPropertyName

Success
Success
Success
Success

YES.
Help Stop-Service –Parameter Name

-Name <String[]>
Specifies the service names of the services to be stopped. Wildcards are permitted.

The parameter name is optional. You can use "Name" or its alias, "ServiceName", or you can omit the parameter name.

Required? true
Position? 1
Default value
Accept pipeline input? true (ByPropertyName, ByValue)
Accept wildcard characters? true

```
Get-Help Start-Service -Parameter NAME

-Name <String[]>
```

“Spooler”,”WSearch” | Start-Service

Task 9.4.2: Modifying Parameter binding with Select-Object

1. We want to list the state of a service on a number of remote machines. Does the following command work?

```
"2012R2-DC","2012R2-MS" | Get-Service
```

Yes \ No

NO.

2. Execute the command and read the error messages displayed.
To which parameter do you think the Get-Service Cmdlet is trying to bind the pipeline input strings?

-Name

3. Get help for Get-Service Cmdlet.
To which parameter s***hould*** the pipeline input objects be bound?

-ComputerName

4. To overcome this limitation, we can use the Select-Object cmdlet to create a new object with a "ComputerName" parameter to store the computer name strings.
The syntax requires a hashtable syntax (@{} - covered in Module 15) to hold the new property name. Execute the command below.

```
"2012R2-DC","2012R2-MS" |
Select-Object @{Name="ComputerName";Expression={$_}}
```

5. Now pipe the command to Get-Member. What Property name is displayed?

ComputerName

6. The pipeline input values have been bound correctly to the –ComputerName parameter on the Get-Service Cmdlet. We can pipe the objects to the Get-Service Cmdlet and list a named service on both remote machines.
We are now binding the incoming object parameter name, to the downstream Cmdlet parameter with the same name, rather than by the incoming object's value.
Type and execute the command below.

Make sure people comprehend this!

```
"2012R2-DC","2012R2-MS" |
Select-Object @{name="ComputerName";Expression={$_}} |
Get-Service -Name BITS |
Format-Table MachineName, Name, Status
```

Task 9.4.3: Modifying Parameter binding with Scriptblocks

Returning to our user creation scenario, some parameters of the New-ADUser Cmdlet can be populated automatically by taking advantage of property name binding (By PropertyName).
This binding behavior allows you to pipe the output from the Import-Clixml Cmdlet directly to the New-ADUser Cmdlet without using a Foreach-Object Cmdlet.

Property name binding only works if the property names on the source object (on the left hand-side of the pipeline character) match the parameter names of the destination Cmdlet (on the Right-hand side). Object properties that do not have a matching parameter name cannot be bound in this way. As we have seen, they ***can*** be bound manually by using Select-Object to add properties with matching names to the source object.

In this exercise we cover another, simpler, way to adapt the binding process, by employing script blocks.

1. First, delete the users and OUs created in the previous exercise, using the command below.

```
Get-ADOrganizationalUnit -Identity "OU=Personnel,DC=contoso,DC=com" |
Set-ADOrganizationalUnit -ProtectedFromAccidentalDeletion:$false -PassThru |
Remove-ADOrganizationalUnit -Recursive -Confirm:$false
```

2. Recreate the root OU by typing the following commands.

```
New-ADOrganizationalUnit -Path "DC=Contoso,DC=com" -Name Personnel
```

3. Next, create the four child OUs.
   Notice that we no longer need a Foreach-Object cmdlet. The New-ADOrganizationalUnit Cmdlet will accept pipeline input for the -Name parameter, so we could have used the Select-Object Cmdlet to bind the input strings by property name. A somewhat simpler approach is to pass the –Name parameter the current pipeline object variable, within curly braces (a scriptblock).
   Each string passed into the pipeline is bound to the New-OrganizationalUnit Cmdlet's –Name parameter.

```
"Engineering","Finance","Marketing","IT" |
New-ADOrganizationalUnit -Path "OU=Personnel,DC=Contoso,DC=com" –Name {$_}
```

4. Examine the object members when C:\PShell\Labs\Lab_9\Contoso-UserImport.xml is imported.

```
Import-Clixml C:\PShell\Labs\Lab_9\Contoso-UserImport.xml |
Get-Member –MemberType Properties
```

5. Then, look at the New-ADUser Cmdlet's parameters in the Windows PowerShell ISE.
   Type the following and wait for intellisense to populate the parameter list.

```
New-ADUser -
```

6. The table below displays property names imported from the xml file in the left-hand column and the parameter names that the New-ADUser Cmdlet implements in the right-hand column.
   Property names that match parameter names will be bound automatically. However, the non-matching property names will need to be bound manually using either the Select-Object Cmdlet, or a scriptblock.
   Identify the XML property names that do not match the New-ADUser Cmdlet's parameter names by circling them. These parameters have to be bound manually.

XML File property Name	New-ADUser Parameter Name
Firstname	GivenName
Lastname	Surname
Name	Name
SamAccountName	SamAccountName
Country	Country
Company	Company
LogonScript	ScriptPath
AccountExpirationDate	AccountExpirationDate
Department	Department
City	City
Address	StreetAddresss
State	State
PostCode	PostalCode
Division	Division
EmployeeID	EmployeeNumber

7. Open the following script in the Windows PowerShell ISE script pane. C:\PShell\Labs\Lab_9\Create-NewUsers.ps1. The script body is shown below.

NOTE: Active Directory user accounts are disabled by default when first created. When enabling a user account you are first required to set a password, which must be passed to the –AccountPassword parameter as a secure string object.

```
$password = ConvertTo-SecureString -String 'PowerShell4' -AsPlainText -Force

Import-CliXml -Path C:\PShell\Labs\Lab_9\Contoso-UserImport.xml |
New-ADUser -Path {"OU=$($_.Department),OU=Personnel,DC=Contoso,DC=com"} `
-GivenName {$_.Firstname} -Surname {$_.Lastname} `
-ScriptPath {$_.LogonScript} -StreetAddress {$_.Address} `
-PostalCode {$_.PostCode} -EmployeeNumber {$_.EmployeeID} `
-DisplayName {"$($_.FirstName) $($_.LastName)"} `
-AccountPassword $password -Enabled $true -PassThru
```

8. Run the script to create the user accounts.

NOTE: The (`) character is the 'backtick' character – the Windows PowerShell escape character. In this instance, it is used to escape the special meaning of a carriage return (command execution separator), allowing a single command to be continued on the next line.