Scripting Techniques

Jeff Hicks
AUTHOR ~ TEACHER ~ SENSEI
@jeffhicks https://jdhitsolutions.com/blog

No difference between interactive commands and a script

Scripts can make life easier

But scripts can be more complicated

Use PowerShell to simplify the process

If statements

ForEach enumerations

Arrays

Hashtables

Custom objects

Try/Catch error handling

```
If (<some condition is true>) {
 <do something>
}
```

If

Run code or do something IF a condition is true

You will need to know how to use comparison operators

No End If required

```
If (<some condition is true>) {
 <do something>
}
Else
{
 <do something else if it isn't true>
}
```

If

You can include an Else option

PowerShell doesn't care about {} positions

But be consistent

```
If (<some condition is true>) {
 <do something>
}
ElseIf (<some other condition is true>) {
 <do something else if this is true>
Else {
 <do something else if none of the tests are true>
}
```

If

You can include an ElseIf option

- Use as many as you want

PowerShell runs code for first matching condition

Final Else is optional

```
if (Test-Path c:\files\data.txt) {
    $data = Get-Content c:\files\data.txt
}
else {
    Write-Warning "Can't find c:\files.data.txt"
}

#script continues....
```

This expression results in $True or $False

```
$age = 42
if ($age -ge 55) {
    $category = "alpha"
}
elseif ($age -ge 40) {
    $category = "bravo"
}
elseif ($age -ge 25) {
    $category = "gamma"
}
else {
    $category = "omega"
}
```

First true statement “wins”

Only if nothing else is true

Enumeration

ForEach-Obje

ForEach

Slide 9

PM1 is "word-Object" a specific command where "Object" needs to be capitalized? If so then this is fine, if not then PS Titlecase should be "ForEach-object". Also relevant on slides 10, 19, 20, 22.
Paul Madden, 1/29/2018

JH1 This is the official format.
Jeff Hicks, 1/29/2018

$_ represents the current object in the pipeline

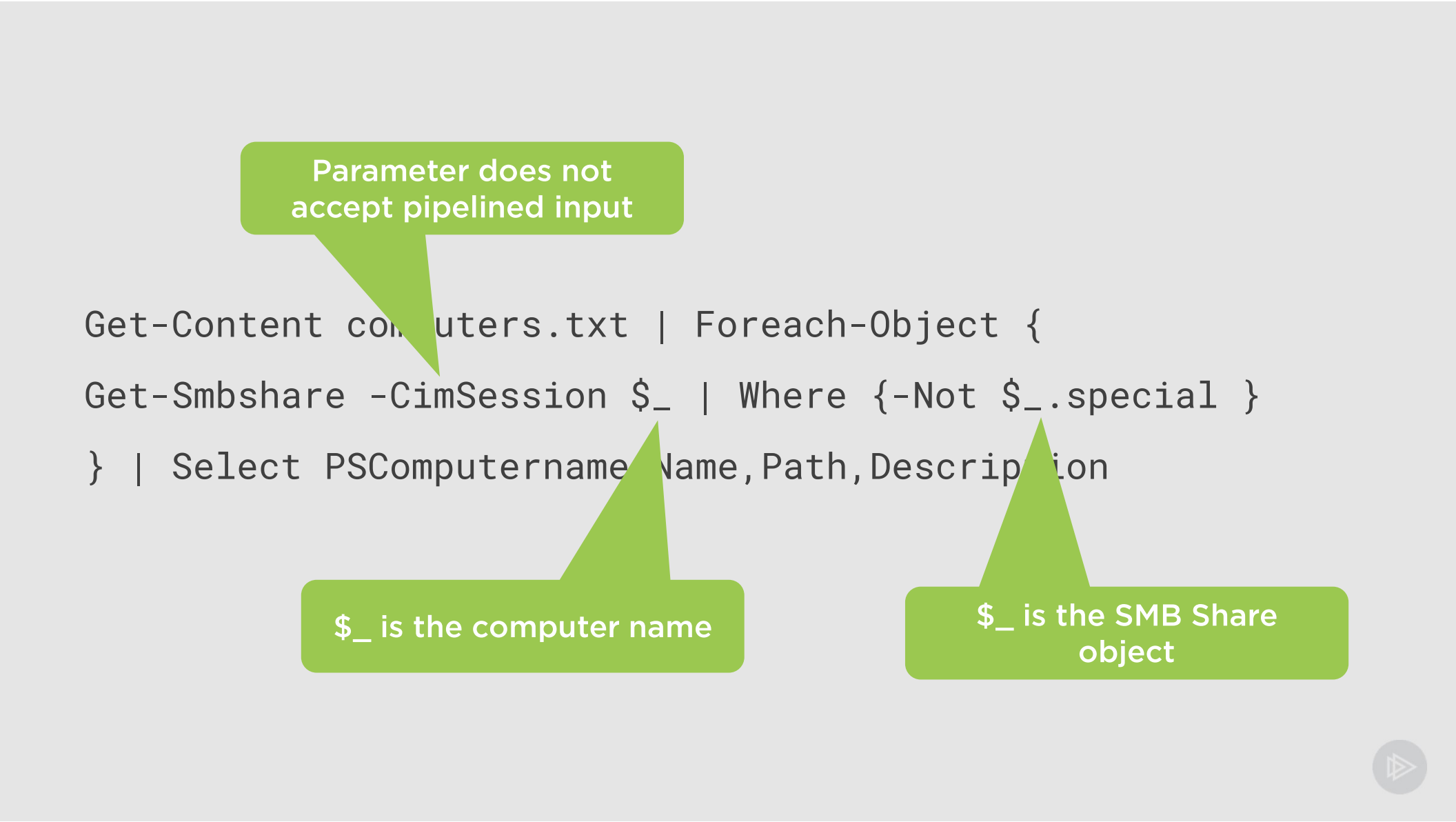
```
2,5,6,8,9 | ForEach { $_ * 3 }
```

ForEach-Object

Has an alias of 'ForEach'

Use when you want to process pipelined objects one at a time

Use when parameter binding won't work


Parameter does not accept pipelined input
Get-Content computers.txt | Foreach-Object {
Get-Smbshare -CimSession $_ | Where {-Not $_.special }
} | Select PSComputername,Name,Path,Description
$_ is the computer name
$_ is the SMB Share object

```
$n = 1..10
foreach ($item in $n) {
    $file = "TestFile-$item.txt"
    New-Item $file
}
```

ForEach Enumerator

“For each something in a group of things, run these commands”

You can use whatever variable you want

Does not pass objects down the pipeline

Arrays

An array is a collection of things

- Created automatically
- $arr = @()

Typically the same object but not required

PowerShell will automatically “unroll” an array

But you can reference items by index number

- 0 based index
- -1 to start at the end

```
PS C:\> $n = 1..5
PS C:\> $n.count
5
PS C:\> $n
1
2
3
4
5
PS C:\> $n[1]
2
PS C:\> $s = Get-Service
PS C:\> $s[-1].Name
ZeroConfigService
PS C:\> help about_arrays
```

- The variable N is an array of numbers
- Count the number of elements in the array
- PowerShell unrolls the array

- Access an item by index number

- Create an array of services
- You can also use an index number starting at the end

- Read the help

```
$arr = @()

$arr += 100


$b = "jeff","jason","don",
"tim“,"adam“

$b –is [array]
```

- Create an empty array
- Add an item to the array
- You can’t remove individual items
- Comma separated items are treated as an array
- Test if something is an array

Hashtable

Collection of Key/Value pairs

AKA dictionary object

Used extensively in PowerShell

```
PS C:\> $hash=@{Name="jeff"}
PS C:\> $hash=@{Name="jeff";
Color="green"; Version =
$PSVersionTable.PSVersion}
PS C:\> $hash

Name           Value
----           -----
Color          green
Name           Jeff
Version        5.1.16299.64
PS C:\> $hash.Add("foo",1)
PS C:\> $hash.color = "red"
```

- @{Key = Value}
- Separate entries with ; or new line
- PowerShell displays entries in an undetermined order
- You can easily modify the hashtable

```
$h = [ordered]@{
  Name = 'Jeff'
  Color = 'green'
  Version = $psversiontable.psversion
}
```

Ordered hashtable

Entries displayed in order entry

Use [ordered] accelerator

Entries added later will be displayed in entry order

```
$params = @{
  Computername = 'Server01'
  Classname = 'win32_logicaldisk'
  Filter = "deviceid='c:'"
  Verbose = $True
}
Get-CimInstance @params
```

Splatting

Define a hashtable of parameter names and values

“Splat” the hashtable to the command

Use the @ symbol to reference the hashtable

Objects in the Pipeline

Select-Object

New-Object

[pscustomobject]

```
dir c:\work -file |
Select-Object Name,LastWriteTime,
@{Name="Size";
Expression={$_.Length}},
@{Name="Age";Expression={(Get-
Date) - $_.lastwritetime}} |
Sort Age -Descending |
Select-Object -first 10
```

Custom object is written to the pipeline

- **Select properties you want**
- **Define new properties with a custom hashtable**
  @{Name='foo';Expression={ <code>} }

  $_ indicates current object

```
$f = dir c:\work -file
$n = Get-Date
foreach ($file in $f) {
 $h=@{
   Name     = $file.name
   Modified = $file.LastWriteTime
   Size     = $file.length
   Age = $n - $file.lastwritetime
 }
  New-Object psobject -Property $h
}
```

- Use whatever enumeration technique works for you
- Define a hashtable of property values
- Write a custom object to the pipeline

```
dir c:\work -file |
foreach-object {
 [pscustomobject]@{
  Name = $_.Name
  Size = $_.length
  Modified = $_.LastWriteTime
  Age = (Get-Date)-$_.LastWriteTime
 }
}
```

Custom object gets written to the pipeline

- Use a type accelerator
- Often easier to read than using Select-Object
- Use whatever enumeration technique works for you

```
Try {
 <some code>
}
Catch {
 <the exception object: $_ >
}
```

Try/Catch

Try to run some command that will create a terminating exception

- -ErrorAction Stop

Catch and handle any errors

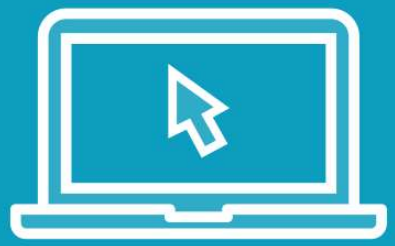
There is an optional Finally block that runs regardless of error

```
$computername = "foo"

Try {

Get-Service bits -computername
$computername -ErrorAction Stop

}

Catch {

Write-Warning "Failed to get
service from $computername.
$($_.exception.message)"

}
```

- Try to run this command
- You must force errors to be ‘terminating’
- Catch errors
- Help about_try_catch_finally

```
WARNING: Failed to get service from foo. Cannot find any service with service name 'bits'.
```

Demo



Scripting Techniques in Action

Summary

No difference between interactive commands and scripting

You can use scripting techniques interactively

More likely to use in a script for more complex operations

Read the about help topics