Learn Windows toolmaking in a month of lunches 读书笔记

Note :
Lab 答案参考 :
https://morelunches.com/2012/12/01/learn-powershell-toolmaking-in-a-month-of-lunches/

Part 1
Introduction to toolmaking

In the PowerShell world, the broadest audience of shell users is just using the tools provided to them. They're running commands and at most combining a bunch of those commands in a script to automate some complex, multistep process.
Toolmakers, on the other hand, are focused less on getting a production task accomplished and more on making a reusable, packaged tool that can complete that task -- and doing so in a way that enables the tool to be handed down to the tool users, who can consume the tool in their own, simpler scripts.

Powershell scripting overview
Install a Virtual machine Windows
Work in Windows PowerShell ISE

PowerShell's scripting language
3.1. One script, one pipeline
COMMAND1;COMMAND2 : one pipeline
COMMAND1
COMMAND2
(two pipelines)

Simple scripts and functions
eg : a simple tool that retrieves some basic operating system information from a remote computer.
Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName DONJONES1D96

ISE或PowerShell命令行中执行多个命令，可用分号隔开 :
eg :
Get-Service;Get-Process

3.2 Variables
Get-Command -noun variable : to see PowerShell commands that work with variables.
eg :
$var = 'hello'
$numbers = 1,2,3,4,5,6

3.3 Quotation Marks
eg :
$name = 'Don'
$prompt = "My name is $name"
eg :
$processes = Get-Process
$prompt = "The first process is using $($processes[0].vm) bytes of VM."
eg :
$debug = "`$computer contains $computer" #the first $ is escaped
$head = "Column`tColumn`tColumn" #`t is the tab character
Note : ` is called backtick or grave accent.
eg : Use double quotes when a string needs to contain single quotes :
$filter1 = "name='BITS'"
$computer = 'BITS'
$filter2 = "name='$computer'"

3.4 Object members and variables

Get-Member : to see the object type name and its members (including properties and methods).
eg :
```
$var = 'Hello'
$var | Get-Member
```
eg :
```
$svc = Get-Service
$svc[0].name  #get the first object's name property
$name = $svc[1].name
$name.length  #get the length property
$name.ToUpper() #invoke the ToUpper method
```
eg :
```
$service = 'bits'
$name = "Service is $service.ToUpper()"
$upper = $name.ToUpper()
$name = "Sergice is $upper"
```

3.5 Parentheses
eg :
```
$name = (Get-Service)[0].name  #$name will contain the name of the first service on the system.
Get-Service - compterName(Get-Content names.txt)  #names.txt contains one computer name per line. Get-Content will return an array of computer names.
```

3.6 Refresher : comparisons
-eq : equal to
-ne : not equal to
-like, -notlike
-gt : greater than
-ge : greater than or equal to
-lt : less than
-le : less than or equal to

3.7 Logical constructs
if construct
switch construct

3.8 Looping constructs
do ... while construct
foreach construct
eg :
```
Get-Service | Stop-Service
```
for construct

3.9 Break and Continue in constructs

4. Simple scripts and functions
4.1 Start with a command
A simple tool that retrieves some basic operating system information from a remote computer :
```
Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName LAPTOP-2BUGV6V3
```

4.2 Turn the command into a script

4.3 Parameterize the command
eg : Parameterizing Get-OSInfo.ps1
```
Param (
    [strng] $computerName = 'localhost'
)
Get-CimInstance -ClassName Win32_OperatingSystem `
                -ComputerName $computerName
```

Note : ` is a 转义字符，使得其后的Return键失效，得以将一行命令输成多行。

运行脚本 ：

```
.\Get-OSInfo -computerName SERVER2
.\Get-OSInfo -comp SERVER2
.\Get-OSInfo SERVER2
.\Get-OSInfo
```

4.4 Turn the script into a function

eg :

Tools.ps1

```
function Get-OSInfo {
    param (
        [string] $computerName = 'localhost'
    )
    Get_CimInstance -ClassName Win32_OperatingSystem `
                    -ComputerName $computername
}
```

4.5 Testing the function

1. Dot sourcing

eg : a dot, a space, then the path and filename to our script.

```
. .\tools.ps1
```

eg :

Dir function: to see the function in memory.

2. Calling the function in the script

eg : revised Tools.ps1

```
function Get-OSInfo {
    param (
        [string[ $computerName = 'localhost'
    )
    Get-CimInstance -ClassName Win32_OperatingSystem `
                    -ComputerName $computerName
}
Get-OSInfo -computername SERVER2
```

3. A better way ahead : modules

4.6 Lab

Using the new CIM cmdlets, write a function to query a computer and find all services by a combination of startup mode such as Auto or Manual and the current state, for example, Running.

eg :

```
Function Get-DiskInfo {
    Param ( [string] $computername='localhost', [int] $MinimumFreePercent=10)
    $disks=Get-WmiObject -Class Win32_Logicaldisk -Filter "Drivetype=3"
    for each ($disk in $disks) {
        $PerFree=($disk.FreeSpace/$disk.Size)*100;
        if ($perFree -ge $MinimumFreePercent) {
            $OK = $TRUE
        }
        else {
            $OK = $FALSE
        };
        $disk | Select DeviceID, VolumeName, Size, FreeSpace, `
        @{ Name = "OK"; Expression = {$OK}}
    }
}
Get-DiskInfo
```

5. Scope (作用域，范围)

5.1 What is scope?

A system of containerization.

There are several elements within PowerShell that are affected by scope :

Variables

Functions

Aliases

PSDrives

PSSnapins

The shell itself is the top-level, or global, scope.

5.2 Seeing scope in action

eg : Script.ps1

```
$var = 'hello!'
function My-Function {
    Write-Host "In the function; var contains '$var'"
    $var = 'goodbye!'
    Write-Host "In the function; var is now '$var'"
}
Write-Host "In the script; var is '$var'"
Write-Host "Running the function"
My-Function
Write-Host "Function is done"
Write-Host "In the script; var is now '$var'"
.\script.ps1
```

5.3 Working out of scope

All of the cmdlets that deal with scoped items have a -scope parameter, which lets you explicitly work with items in a different scope. The arameter takes one of two value types :

A number, o : current scope, 1 : parent scope, 2 : parent's parent scope, and so on

A word, Local : current scope, Script : the script scope that's nearest to you in the hierarchy, Global : refers to the shell's top-level scope.

The default value is 1, or Local.

eg : create a variable named $Color, setting it to Purple, and doing so in your parent script's scope.

```
New-Variable -Name Color -Value Purple Scope 1
```

eg : create or set a global variable named $color, setting it to contain purple, no matter where this command was executed.

```
$global:color = 'purple'
```

5.4 Getting strict with scope

Set-StrictMode : configures some options that affect how scope works.

5.5 Best practices for scope

5.6 Lab

eg : to create some PSDrives based on environmental variables like %APPDATA% and %USERPROFILE%\DOCUMENTS.

```
Function New-Drives {
    Param ()
    New-PSDrive -Name AppData -PSProvider FileSystem -Root $env:Appdata
    New-PSDrive -Name Temp -PSProvider FileSystem -Root $env:TEMP
    $mydocs=Join-Path -Path $env:userprofile -ChildPath Documents
    New-PSDrive -Name Docs -PSProvider FileSystem -Root $mydocs
}
New-Drives
DIR temp: | measure-object -property length -sum
```

Part 2 Building an inventory tool

6 Tool design guidelines

6.1 Do one thing, and do it well

A function should do one -- and only one -- of these things :

Retrieve data from someplace

Process data

Output data to some place

Put data into some visual format meant for human consumption

1. Input tools

Restart-Computer

Get-EventLog

Get-Process

Invoke-Command

Import-CSV

Get-Content

Get-ADComputer

2. Functional tools

3. Output tools

6.2 Labs

1. Design a command that will retrieve the following information from one or more remote computers, using the indicated WMI classes and properties :

Win32_ComputerSystem:

Workgroup

AdminPasswordStatus; display the numeric values of this property as text strings

For 1, display Disabled

For 2, display Enabled

For 3, display NA

For 4, display Unknown

Model

Manufacturer

From Win32_BIOS

SerialNumber

From Win32_OperatingSystem

Version

ServicePackMajorVersion

Your function's output should also include each computer's name.

Ensure that your function's design includes a way to log errors to a text file, allowing the user to specify an error filename but defaulting to C:\Errors.txt. Also plan ahead to create a custom view so that your function always outputs a table, using the following column headers :

ComputerName

Workgroup

AdminPassword

Model

Manufacturer

BIOSSerial

OSVersion

SPVersion

2. Design a tool that will retrieve the WMI Win32_Volume class from one or more remote computers. For each computer and volume, the function should output the computer's name, the volume name (such as C:|), and the volume's free space and size in GB (using no more than two decimal places). Only include volumes that represent fixed hard drives -- don't include optical or network drives in the output. Keep in mind that any give computer may have multiple hard disks; your function's output should include one object for each disk.

Ensure that your function's design includes a way to log errors to a text file, allowing the user to specify an error filename but defaulting to C:\Errors.txt. Also plan ahead to create a custom view so that your function always outputs a table, using the following column headers :

ComputerName

Drive

FreeSpace

Size

3. Design a command that will retrieve all running services on one or more remote computers. This command will offer the option to log the names of failed computers to a text file. It will produce a list that includes each running service's name and display name, along with information about the process that represents each running service. That information will include the process name, virtual memory size, peak page file usage, and thread count. But peak page file usage and thread count will not display be default.

7 Advanced functions, part 1

7.1 Advanced function template

```
function <name> {
    [CmdletBinding () ]
    param (
    )
    BEGIN { }
    PROCESS { }
    END { }
}
```

7.2 Designing the function

The properties of the output object are as follows :

ComputerName : the name of the computer

OSVersion : the Windows version

SPVersion : the service pack version

BIOSSerial : the BIOS serial number

Manufacturer : the computer's manufacturer

Model : the computer's model description

We want to be able to specify one or more computer names via parameter :

Get-SystemInfo -computer one, two, three -eeorlog retries.txt

We'd also like to be able to pipe in computer names.

eg :

Get-Content computers.txt | Get-SystemInfo -errorlog retries.txt

7.3 Declaring parameters

eg : Addng parameters to our script

```
function <name> {
    [CmdletBinding ()]
    param (
        [string[]] $ComputerName,
        [string] $ErrorLog
    )
    BEGIN { }
    PROCESS { }
    END { }
}
```

7.4 Testing the parameters

eg : Adding throwaway code to the function

```
function Get-SystemInfo {
    [CmdletBinding () ]
    param (
        [string[]] $ComputerName,
        [string] $ErrorLog
    )
    BEGIN { }
    PROCESS {
        Write-Output $ComputerName
        Write-Output $ErrorLog
    }
```

```
    END { }
}
Get-SystemInfo -computername one, two, three -errorlog x.txt
Get-SystemInfo one x.txt
.\test/ps1
eg : Removing the throwaway code
function Get-SystemInfo {
    [CmdletBinding () ]
    param (
        [string[]] $ComputerName,
        [string] $ErrorLog
    }
    BEGIN { }
    PROCESS {
    }
    END { }
}
```

7.5 Writing the main code

eg : Beginning the functional code

```
function Get-SystemInfo {
    [CmdletBinding () ]
    param (
        [string[]] $ComputerName
        [string] $ErrorLog
    }
    BEGIN {
        Write-Output "Log name is $errorlog"
    }
    PROCESS {
        foreach ($computer in $computername) {
            Write-Output "computer name is $computer"
        }
    }
    END { }
}
Get-SystemInfo -ComputerName one, two, three -ErrorLog x.txt
```

eg : Adding the WMI commands

```
function Get-SystemInfo {
    [CmdletBinding () ]
    param (
        [string[]] $ComputerName,
        [string] $ErrorLog
    )
    BEGIN {
        Write-Output "Log name is $errorlog"
    }
    PROCESS {
        foreach ($computer in $computername) {
            $os = Get-WmiObject -class Win32_OperatingSystem `
                                -computerName $computer
            $comp = Get-WmiObject -class Win32_ComputerSystem `
                                  -computerName $computer
            $bios = Get-WmiObject -class Win32_BIOS `
                                  -computerName $computer
        }
    }
    END { }
```

```
}
```

7.6 Outputting custom objects

eg : Creating the custom output

```
function Get-SystemInfo {
    [CmdletBinding () ]
    param (
        [string[]] $ComputerName,
        [string] $ErrorLog
    }
    BEGIN {
        Write-Output "Log name is $errorlog"
    }
    PROCESS {
        foreach ($computer in $computername) {
            $os = Get-WmiObject -class Win32_OperatingSystem `
                                -computerName $computer
            $comp = Get-WmiObject -class Win32_ComputerSystem `
                                    -computername $computer
            $bios = Get-WmiObject -class Win32_BIOS `
                                   -computerName $computer
            $props = @{'ComputerName' = $computer;
                        'OSVersion' = $os.version;
                        'SPVersion' = $os.servicepackmajorversion;
                        'BIOSSerial' = $bios.serialnumber;
                        'Manufacturer' = $comp.manufacturer;
                        'Model' = $comp.model}
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
    END { }
}
Get-SystemInfo -ErrorLog x.txt -ComputerName localhost, localhost
```

eg :

```
Get-SystemInfo -comp localhost -errorlog x.txt | Export-CSV
Get-SystemInfo -comp localhost -errorlog x.txt | ConvertTo-HEML
Get-SystemInfo -comp localhost -errorlog x.txt | Export-CliXML
Get-SystemInfo -comp localhost -errorlog x.txt | Sort OSVersion
Get-SystemInfo -comp localhost -errorlog x.txt | Format-Table
```

7.7 What ot to do

eg : Removing the last line of throwaway code

```
function Get-SystemInfo {
    [CmdletBinding () ]
    param (
        [string[]] $computerName,
        [string] $ErrorLog
    }
    BEGIN {
    }
    PROCESS {
        foreach ($computer in $computername) {
            $os = Get-WmiObject -class Win32_OperatingSystem `
                                -computerName $computer
            $comp = Get-WmiObject -class Win32_ComputerSystem `
                                    -computerName $computer
            $bios = Get-WmiObject -class Win32_BIOS `
```

```
                                -computerName $computer
            $props = @{'ComputerName' = $computer;
                       'OSVersion' = $os.version;
                       'SPVersion' = $os.ervicepackmajorversion;
                       'BIOSSerial' = $bios.serialnumber;
                       'Manufacturer' = $comp.manufacturer;
                       'Model' = $comp.model }
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
    END { }
}
Get-SystemInfo -ErrorLog x.txt -ComputerName localhost, localhost
```

7.9 Labs

1 Lab A

Write an advanced function that accepts one or more computer names. For each computer name, use CIM or WMI to query the specified information. For now, keep each property's name, using ServicePackMajorVersion, Version, SerialNumber, and so on.

Test the function by adding <function-name> -computerName localhost to the bottom of your script and then running the script (replacing <function_name> with your actual function name, which would not include the angle brackets). The output for a single service should look something like this:

2. Lab B

Using your notes for Lab B from chapter 6, write an advanced function that accepts one or more computer names. For each computer name, use CIM or WMI to query the specified information. Format the Size and FreeSpace property values in GB to two decimal points. Test the function by adding <function-name> -computerName localhost to the bottom of your script and then running the script (replacing <function_name> with your actual function name, which would not include the angle brackets). The output for a single service should look something like this:

3 Lab C

Using your notes for Lab C from chapter 6, write an advanced function that accepts one or more computer names. For each computer name, use CIM or WMI to query all instances of Win32_Service where the State property is Running. For each service, get the ProcessID property. Then query the matching instance of the Win32_Process class -- that is, the instance with the same ProcessID. Write a custom object to the pipeline that includes the service name and display name, the computer name, the process name, ID, virtual size, peak page file usage, and thread count. Test the function by adding <function_name> -computerName localhost to the end of the script (replacing <function_name> with your actual function name, which would not include the angle brackets). The output for a single service should look something like this:

4. Standalone lab

Write an advanced function named Get-SystemInfo. This function should accept one or more computer name via a -ComputerName parameter. It should then use WMI or CIM to query the Win32_OperatingSystem class and Win32_ComputerSystem class for each computer. For each computer queried. display the last boot time (in a standard date/time format), the computer name, and operating system version (all from Win32_OperatingSystem). Also, display the manufacturer and model (from Win32_ComputerSystem). You should end up with a single object with all of this information for each computer.

Note that the last boot time property does not contain a human-readable date/time value; you'll need to use the class's ConvertToDateTime() method to convert that value to a normal-looking date/time. Test the function by adding Get-SystemInfo -computerName localhost to the end of the script.

You should get a result like this :

8. Advanced functions, part 2

8.1 Making parameters mandatory

eg : Adding parameter attributes

```
function Get-SystemInfo {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory=$True)]
        [string[]] $ComputerName,
        [string] $ErrorLog = 'c:\retry.txt'
```

```
    )
    BEGIN {
    }
    PROCESS {
        foreach ($computer in $computername) {
        $os = Get-WmiObject -class Win32_OperatingSystem `
                            -computerName $computer
        $comp = Get-WmiObject -class Win32_ComputerSystem `
                                -computerName $computer
        $bios = Get-WmiObject -class Win32_BIOS `
                              -computerName $computer
        $props = @{'ComputerName'=$computer;
                   'OSVersion' = $os.version;
                   'SPVersion' =  $os.servicepackmajorversion;
                   'BIOSSerial' = $bios.serialnumber;
                   'Manufacturer' = $comp.manufacturer;
                   'Model' = $comp.model}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
    }
}
END {}
}
Get-SystemInfo
```

8.2 Verbose output

eg : Adding verbose output

```
Write-Verbose "Error log will be $ErrorLog"
Write-Verbose "Querying $computer"
Write-Verbose "WMI queries complete"
```

8.3 Parameter aliases

```
param (
    [Parameter (Mandatory = $True)]
    [Alias ('hostname')]
)
```

8.4 Accepting pipeline input

eg : configuring -ComputerName to accept pipeline input

```
[Parameter(Mandatory=$True, ValueFromPipeline = $True)]
```

8.5 Parameter validation

add a validation attribute so that PowerShell will only accept 1 to 10 computer names.

eg : Adding a validation attribute to -ComputerName

```
param (
    [Parameter(Mandatory=$True, ValueFromPipeline = $True)]
    [ValidateCount(1,10)]
)
```

8.6 Adding a switch parameter

eg : Adding a switch parameter

```
param (
    [Parameter(Mandatory = $True, ValueFromPipeline = $True)]
    [ValidateCount(1,10)]
    [Alias('hostname')]
    [ string[]] $ComputerName,
    [string] $ErrorLog = 'c:\retry.txt',
    [switch] $LogErrors
```

)

PowerShell will automatically populate it with True if the command is run with -LogErrors and populate it with False if the command is run without the parameter.

8.7 Parameter help

add some parameter help. This will help folks understand what each parameter is meant to do, especially the -ComputerName parametr, which is something they may be prompted for.

eg :

```
function Get-SystemInfo {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $True,
                   ValueFromPipeline = $True,
                   HelpMessage = "Computer name or IP address")]
        [ValidateCount(1,10)]
        [Alias('hostname')]
        [string[]] $ComputerName,
        [string] $ErrorLog = 'c:\retry.txt',
        [switch] $LogErrors
    )
    BEGN {
        Write-Verbose "Error log will be $ErrorLog"
    }
    PROCESS {
        Write-Verbose "Beginning PROCESS block"
        foreach ($computer in $computername) {
            Write-Verbose "Querying $computer"
            $os = Get-WmiObject -class Win32_OperatingSystem -computerName $computer
            $comp = Get-WmiObject -class Win32_ComputerSystem -computerName $computer
            $bios = Get-WmiObject -class Win32_BIOS -computerName $computer
            $props = @{'ComputerName' = $computer;
                       'OSVersion' = $os.version;
                       'SPVersion' = $os.servicepackmajorversion;
                       'BIOSSerial' = $bios.serialnumber;
                       'Manufacturer' = $comp.manufacturer;
                       'Model' = $comp.model}
            Write-Verbose "WMI queries complete"
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
    END {}
}
Get-SystemInfo
```

8.9 Labs

1. Lab A

Modify your advanced function from chapter 7, Lab A, to accept pipeline input for the -ComputerName parameter. Also, add verbose input that will display the name of each computer contacted. Include code to verify that the -ComputerName parameter will not accept a null or empty value. Test the function by adding 'localhost' | <function-name> -verbose to the end of your script. The output should look something like this :

2. Lab B

Modify your advanced function from chapter 7, Lab B, to accept pipeline input for the -ComputerName parameter. Add verbose output that will display the name of each computer contacted. Ensure that the -ComputerName parameter will not accept a null or empty value. Test the function by adding 'localhost' | <function-name> -verbose to the end of your script. The output should look something like this :

3. Lab C

Modify your advanced function from Lab C in chapter 7 to accept pipeline input for the -ComputerName parameter. Add verbose

output that will display the name of each computer contacted and the name of each service queried. Ensure that the -ComputerName parameter will not accept a null or empty value. Test the function by running 'localhost' | <function-name> -verbose. The output for two services should look something like this :

4. Standalone lab

Use this script as your starting point.

eg : Standalone script

```
function Get-SystemInfo {
    [CmdletBinding()]
    param (
        [string[]] $ComputerName
    )
    PROCESS {
        foreach ($computer in $computerName) {
            $os = Get-WmiObject -class Win32_OperatingSystem -computerName $computer
            $cs = Get-WmiObject -class Win32_ComputerSystem -computerName $computer
            $props = @{'ComputerName' = $computer;
                    'LastBootTime' = ($os.ConvertToDateTime($os.LastBootupTime));
                    'OSVersion' = $os.version;
                    'Manufacturer' = $cs.manufacturer;
                    'Model' = $cs.model}
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
}
```

Modify this function to accept pipeline input for the ComputerName parameter. Add verbose output that will display the name of each computer contacted. Ensure that the -ComputerName parameter will not accept a null or empty value. Test the script by adding this line to the end of the script file :

```
'localhost','localhost' | Get-SystemInfo -verbose
```

The output should look something like this :

9 Writing help

Have your tolls include help that looks just like the help for PowerShell's native cmdlets.

9.1 Comment-based help

eg : Adding comment-based help to our function

```
function Get-SystemInfo {
<#
.SYNOPSIS
Retrieves key system version and model information from one to ten computers.
.DESCRIPTION
Get-SystemInfo uses Windows Management Instrumentation
(WMI) to retrieve information from one or more computers.
Specify computers by name or by IP address.
.PARAMETER ComputerName
One or more computer name or IP addresses, up to a maximum of 10.
.PARAMETER LogErrors
Specify this switch to create a text log file of computers that could not be queried.
.PARAMETER ErrorLog
When used with -LogErrors, specifies the file path and name
to which failed computer name will be written. Defaults to
C:\Retry.txt
.EXAMPLE
Get-Content names.txt | Get-SystemInfo
.EXAMPLE
Get-SystemInfo -ComputerName SERVER1, SERVER2
#>
    [CmdletBinding()]
```

```
    param (
        [Parameter (Mandatory = $True,
                    ValueFromPipeline = $True,
                    Helpmessage = "Computer name or IP address")]
        [ValidateCount(1,10)]
        [Alias('hostname')]
        [string[]] $ComputerName,
        [string] $ErrorLog = 'c:\retry.txt',
        [switch] $LogErrors
    )
    BEGIN {
        Write-Verbose "Error log will be $ErrorLog"
    }
    PROCESS {
        Write-Verbose "Beginning PROCESS block"
        foreach ($computer in $computername) {
        Write-Verbose "Querying $computer"
        $os = Get-WmiObject -class Win32_OperatingSystem -computerName $computer
        $comp = Get-WmiObject -class Win32_ComputerSystem -computerName $computer
        $bios = Get-WmiObject -class Win32_BIOS -computerName $computer
        $props = @{'ComputerName' = $computer;
                    'OSVersion' = $os.version;
                    'SPVersion' = $os.servicepackmajorversion;
                    'BIOSSerial' = $bios.serialnumber;
                    'Manufacturer' = $comp.manufacturer;
                    'Model' = $comp.model}
        Write-Verbose "WMI queries complete"
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
    }
}
END {}
}
help Get-SystemInfo -full
```

9.2 XML-based help

Lets you provide help in multiple languages.

9.4 Labs

1. Lab A

Add comment-based help to your advanced function from Lab A in chapter 8, Include at least a synopsis, description, and help for the -ComputerName parameter. Test your help by adding help <function-name> to the end of your script.

2. Lab B

Add comment-based help to your advanced function from Lab B in chapter 8. Include at least a synopsis, description, and help for the -ComputerNAame parameter. Test your help by adding help <function-name> to the end of your script.

3. Lab C

Add comment-based help to your advanced function from Lab C in chapter 8. Include at least a synopsis, description, and help for the -ComputerName parameter. Test your help by adding help <function-name> to the end of your script.

4. Standalone lab

Using the script in the following listing, add comment-based help.

10 Error handling

10.1 It's all about the action

Whenever a PowerShell command -- be it a native cmdlet or a function you write -- encounters a non-terminating error, it asks PowerShell what to do. PowerShell looks at a built-in variable, $ErrorActionPreference, to see what it should do.

Non-terminating error : any error that presents a problem, but one from which the command can recover and continue.

$ErrorActionPreference four values :

Continue : the default value,

SilentlyContinue : just shutup and get on with it.
Stop : turns the non-terminating error into a terminating exception, meaning the comamdn stops.
Inquire : "ask me what to do." Literally, with a prompt.

10.2 Setting the error action

10.3 Saving the Error
-ErrorVariable or -EV : lets you specify a variable name, and any error produced by the command will be stored in that variable so that you can examine it and take whatever action you like.
eg : Get-WmiObject -Class Win32 BIOS -ComputerName NOTONLINE -EV err -EA SlientlyContinue
$err
eg : You have to be really careful with -ErrorVariable. The error is not $x, the error is $fred.
$x='fred'
Gwmi Win32_BIOS -ErrorVariable $x
eg : If you intended for the error to go into $x, then
Gwmi Win32_BIOS -ErrorVariable x

10.4 Error handling v1: Trap
For further information :
help about_trap

10.5 Error Handling : Try... Catch... Finally
eg :

```
function Get-SystemInfo
<#
.SYNOPSIS
Retrieves key system version and model information
from one to ten computers.
.DESCRIPTION
Get-SystemInfo uses Windows Management Instrumentation
(WMI) to retrieve information from one or more computers.
Specify computers by name or by IP address.
.PARAMETER ComputerName
One or more computer names or IP addresses, up to a maximum of 10.
.PARAMETER LogErrors
Specify this switch to create a text log file of computers
that could not be queried.
.PARAMETER ErrorLog
When used with -LogErrors, specifies the file path and name to which failed computer names will be written. Defaults to
C:\Retry.txt.
.EXAMPLE
Get-Content names.txt | Get-SystemInfo
.EXAMPLE
Get-SystemInfo -ComputerName SERVER1, SERVER2
#?
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True, ValueFromPipeline=$True,
        HelpMessage='Computer name or IP address")]
        [ValidateCount(1,10)]
        [Alias('hostname')]
        [String[] $ComputerName,
        [string] $ErrorLog = 'c:\retry.txt',
        [switch] $LogErrors
    )
    BEGIN {
        Write-Verbose "Error log will be $ErrorLog"
    }
```

```
PROCESS {
    Write-Verbose "Beginning PROCESS block"
    foreach ($computer in $computername) {
        Write-Verbose "Querying $computer"
        Try {
            $os = Get-WmiObject -class Win32_OperatingSystem `
                                -computerName $computer `
                                -erroraction Stop
        } Catch {
            if ($LogErrors) {
                $computer | Out-File $ErrorLog -Append
            }
        }
        $comp = Get-WmiObject -class Win32_ComputerSystem `
                               -computerName $computer
        $bios = Get-Wmiobject -class Win32_BIOS `
                              -computerNme $computer
        $props = @{'ComputerName'=$computer;
                   'OSVersion'=$os.version;
                   'SPVersion'=$os.servicepackmajorversion;
                   'BIOSSerial'=$bios.serialnumber;
                   'Manufacturer'=$comp.manufacturer;
                   'Model'=$comp.model}
        Write-Verbose "WMI queries complete"
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
    }
}
END{}
}
Get-SystemInfo -computername NOTONLINE -logerrors
```

10.6 Providing some visuals

eg : displays a warn (which is less severe than an error) to the user.

```
function Get-SystemInfo {
<#
.SYNOPSIS
Retrieves key system version and model information
from one to ten computers.
.DESCRIPTION
Get-SystemInfo uses Wondows Management Instrumentation (WMI)
to retrieve information from one or more computers.
Specify computers by name or by IP address.
.PARAMETER ComputerName
One or more computer names or IP addresses, up to a maximum of 10.
.PARAMETER LogErrors
Specify this switch to create a text log file of computers that could not be queried.
.PARAMETER ErrorLog
When used with -LogErrors, specifies the file path and name to which failed computer
names will be written. Defaults to C:\Retry.txt.
.EXAMPLE
Get-Content names.txt | Get-SystemInfo
.EXAMPLE
Get-SystemInfo -ComputerName SERVER1, SERVER2
#>
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,
```

```
                    ValueFromPipeline=$True,
                    HelpMessage="Computer name or IP address")]
        [ValidateCount(1,10)]
        [Alias('hostname')]
        [string[] $ComputerName,
        [string] $ErrorLog = 'c:\retry.txt',
        [switch] $LogErrors
    )
    BEGIN {
        Write-Verbose "Error log wll be $ErrorLog"
    }
    PROCESS {
        Write-Verbose "Beginning PROCESS block"
        foreach ($computer in $computername) {
            Write-Verbose "Querying $computer"
            Try {
                $everything_ok = $true
                $os = Get-WmiObject -class Win32_OperatingSystem -computerName $computer -erroraction Stop
            } Catch {
                $everything_ok = $false
                Write-Warning "$computer failed"
                if ($LogErrors) {
                    $computer | Out-File $ErrorLog -Append
                    Write-Warning "Logged to $ErrorLog"
                }
            }
            if (everything_ok) {
                $comp = Get-WmiObject -class Win32_ComputerSystem -computerName $computer
                $bios = Get-WmiObject -class Win32_BIOS -computerName $computer
                $props = @{'ComputerName'=$computer;
                            'OSVersion'=$os.version;
                            'SPVersion'='$os.servicepackmajorversion;
                            'BIOSSerial'=$bios.serialnumber;
                            'Manufacturer'=$comp.manufacturer;
                            'Model'=$comp.model}
                Write-Verbose "WMI queries complete"
                $obj = New-Object -TypeName PSObject -Property $props
                Write-Output $obj
            }
        }
    }
    end { }
}
Get-SystemInfo -ComputerName NOTONLINE -LogErrors
eg : call the commandline
PS C:\> C:\test.ps1
```

11. Debuggin g techniques

11.1 Two types of bugs

Typos & logic errors

11.2 Solving typos

1) script editing software

2) format your scripts

3) read error messages

11.3 The real trick to debugging : expectations

11.4 Dealing with logic errors : trace code

By using Write-Debug

eg :

```
[DmdletBinding()]
param()
$data = import-csv c:\data.csv
Write-Debug "Imported CSV data"

$totalqty = 0
$totalsold = 0
$totalbought = 0
foreach ($line in $data) {
    if ($line.transaction -eq 'buy') {
        Write-Debug "ENDED BUY transaction (we sold)"
        $totalqty -= $line.qty
        $totalsold = $line.total
    } else {
        $totalqty += $line.qty
        $totalsold = $line.total
    } else {
        $totalqty += $line.qty
        $totalbought = $line.total
        Write-Debug "ENDED SELL transaction (we bought)"
    }
}
Wrtie-Debug "OUTPUT: $totalqty,$totalbought, $totalsold, $($totalbought-$totalsold)"
```

11.5 Dealing with logic errors : breakpoints

By running Set-PSBreakpoint.

F9 (to set line-based breakpoints in the ISE)

eg :

```
Set-PSBreakpoint -Script C:\debug.ps1 -Variable totalbought, totalsold -Mode ReadWrite
```

11.6 Seriously, have expectations

11.8 Labs

12 Creating custom format views

12.1 The anatomy of a view

PowerShell ships with a number of views, all of which are contained in .format.ps1xml files that live within PowerShell's installation folder.

cd $pshome : change directory to installation home of PowerShell's installation folder.

12.2 Adding a type name to output objects

12.3 Making a view

12.4 Loading and debugging the view

Update-FormatData : load view files into memory within each new shell session.

eg :

```
Update-FormatData -PrependPath C:\test.format.ps1xml
```

12.5 Using the view

12.7 Labs

13 Script and manifest modules

13.1 Introducing modules

13.1.1 Module loation

get-content env:\psmodulepath

You can modify this environment variable using either Windows or a Group Policy Object (GPO) to contain additional paths.

eg : to create the necessary path :

New-Item -type directory -path (((get-content env:\psmodulepath) -split ';') [0])

13.1.2 Module name

eg :

cd .\users\donjones\Documents\WindowsPowerShell\Modules

mkdir

MOLTools

13.1.3 Module contents

Import-Module MOLTools : to load this module

order of automatically loading a module :

1) a module manifest (eg. MOLTools.psd1)

2) a binary module (eg. MOLTools.dll)

3) a script module (eg. MOLTools.psm1)

13.2 Creating a script module

13.3 Creating a module manifest

13.4 Creating a module-level setting variable

13.6 Labs

14 Adding database access

14.1 Simplifying database access

low-level .NET Framework technology

14.2 Setting up your environment

eg : check if you have a SQL Server service running

get-service -name mssql* | select name

14.3 The database functions

14.4 About the database functions

Get-MOLDatabaseData : to query information from a database

Invoke-MOLDatabaseQuery : to make changes, such as adding data, removing data, or changing data.

Each supports three parameters

-ConnectionString

-isSQLServer

-Query

Note : Invoke-MOLDatabaseQuery doesn't write anything to the pipeline; it just runs your query. Get-MOLDatabaseData will retrieve data and place it into the pipeline.

14.5 Using the database functions

15 Interlude : creating anew tool

15.1 Designing the tool

Write a function named Get-RemoteSmShare.
It should accept one or more computer names, either on a -ComputerName parameter or from the pipeline, and then retrieve a list of current shared folders from each specified computer. The output must include each computer's name, the share name, description, and the path to the share.

15.2 Writing and testing the function
test :
Get-RemoteSmbShare -computerName localhost, localhost

15.3 Dressing up the parameters
include the following features :
The -ComputerName parameter should be mandatory, meaning PowerShell should prompt for a value if one isn't specified.
The -ComputerName parameter should accept input from the pipeline ByValue.
Add -HostName as an alias for the -ComputerName parameter.
Ensure that at least one, and no more than five, computer names are specified each time the function is run.

15.4 Adding help
test :
Help Get-RemoteSmbShare

15.5 Handling errors
Modify your function to include a -ErrorFile parameter.
This parameter should accept a single string and should default to C:\Errors.txt.
Modify the function to catch any errors that occur while running Invoke-Command. When an error occurs, the function should log the failed computer name to whatever file-name is specified in the -ErrorFile parameter. It should always append values to this file and should never attempt to delete the file.
test :
Get-RemoteSmbShare -computer localhost, NOTONLINE, localhost

15.6 Making a module

Part 3 Advanced toolmaking techniques

16 Making tools that make changes

16.1 The -COnfirm and -WhatIf parameters

16.2 Passthrough ShouldProcess
Create a tool called Restart-MOLCimComputer which accept one or more computer names and will utilize WMI to restart them. (using the Reboot() method of the Win32_OperatingSystem class.)
eg :
Invoke-CimMethod -ClassName Win32_OperatingSystem -MethodName Reboot -ComputerName localhost
or
Invoke-WmiMethod -Class Win32_OperatingSystem -Name Reboot -ComputerName localhost
or
enable PowerShell's Remoting by :
Enable -PSRemoting (as an administrator)

16.3 Defining the impact level
PowerShell has two built-in variables
$WhatIfPreference : $False by default
$ConfirmPreference : a command with an impact level of High should always result in a confirmation prompt, unless you run it with -Confirm:$False.

16.4 Implementing ShouldProcess
Create a tool to change a service's logon password.(Can be accomplished by running the Change() method of WMI's Win32_Service class.)

eg :

Get-WmiObject -Class Win32_Service -ComputerName Localhost -Filter "name='BITS'" | Invoke-WmiMethod -Name Change -ArgumentList $null, $null, $null, $null, $null, $null, $null, "P@ssw0rd"

The -ArgumentList parameter of Invoke-WmiMethod can't deal with $null values.

eg :

```
function Set-MOLServicePassword {
[CmdletBinding(SupportsShouldProcess=$True,
ConfirmImpact='Medium')]
param(
[Parameter(Mandatory=$True,
ValueFromPipeline=$True)]
[string[]]$ComputerName,
[Parameter(Mandatory=$True)]
[string]$ServiceName,
[Parameter(Mandatory=$True)]
[string]$NewPassword
)
PROCESS {
foreach ($computer in $computername) {
$svcs = Get-WmiObject -ComputerName $computer `
-Filter "name='$servicename'" `
-Class Win32_Service
foreach ($svc in $svcs) {
if ($psCmdlet.ShouldProcess("$svc on $computer")) {
$svc.Change($null,
$null,
$null,
$null,
$null,
$null,
$null,
$NewPassword) | Out-Null
}
}
}
}
}
Export-ModuleMember -Variable MOLErrorLogPreference
Export-ModuleMember -Function Get-MOLSystemInfo,
Get-MOLComputerNamesFromDatabase,
Set-MOLInventoryInDatabase,
Restart-CimComputer,
Set-MOLServicePassword
```

test :

```
Set-MOLServicePassword -ServiceName BITS -NewPassword "P@ssw0rd" -ComputerName localhost
$ConfirmPreference = "Medium"
Set-MOLServicePassword -ServiceName BITS -NewPassword "P@ssw0rd" -ComputerName localhost
$ConfirmPreference = "High"
Set-MOLServicePassword -ServiceName BITS -NewPassword "P@ssw0rd" -ComputerName localhost -WhatIf
Set-MOLServicePassword -ServiceName BITS -NewPassword "P@ssw0rd" -ComputerName localhost -confirm
```

16.5 Lab

17 Creating a custom type extension

a type extension can actually add functionality to objects you write to the pipeline.

17.1 The anatomy of an extension

eg :

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

17.2 Creating a script property

eg :

```
Get-MOLSystemInfo -ComputerName localhost | Format-List *
```

17.3 Creating a script method

17.4 Loading the extension

eg :

```
Get-MOLSystemInfo -ComputerName localhosst | Get-Member
```

load our ETS XML file into memory :

```
Update-TypeData -PrependPath .\MOLTools.ps1xml
```

17.5 Testing the extension

eg :

```
Get-MOLSystemInfo -ComputerName localhost | Get-Member
```

eg :

```
PS C:\> Get-MOLSystemInfo -ComputerName localhost |
>> Select-Object -Property Com
puterName,OSVersion,NormalizedBIOSSerial |
>> Format-Table -AutoSize
PS C:\> $obj = Get-MOLSystemInfo -ComputerName localhost
PS C:\> $obj.CanPing()
PS C:\> Get-MOLSystemInfo -ComputerName localhost,NOTONLINE |
>> Where-Object -FilterScript { $_.CanPing() }
```

17.6 Adding the extension to a manifest

eg :

```
PS C:\> move .\MOLTools.ps1xml C:\Users\donjones\Documents\WindowsPowerShell\Modules\MOLTools
```

17.7 Lab

18 Creating PowerShell workflows

New feature of PowerShell v3.

18.1 Workflow overview

Workflow include detailed logging and tracking of each and include the ability to retry steps that fail.

Table 18.1 Function or workflow

Function	Workflow
Executed by PowerShell	Executed by workflow engine
Logging and retry attempts through complicated coding	Logging and retry attempts part of the workflow engine
Single-action processing	Supports parallelism
Runs to completion	Can run, pause, and restart
Data loss possible during network problems	Data can persist during network problems
Full language set and syntax	Limited language set and syntax
Runs cmdlets	Runs activities

18.1.1 Common parameters for workflows

built-in common parameters for workflow :

-PSComputerName

-PSParameterCollection

-PSCredential

-PSPersist

-PSPort

-PSUseSSL

-PSSessionOption

18.1.2 Activities and stateless executions

eg1 : workflow with variables

```
Import-Module PSWorkflow
workflow Test-Workflow {
$a = 1
```

```
$a
$a++
$a
$b = $a + 2
$b
}
Test-Workflow
eg2 : workflow that won't work properly
Import-Module PSWorkflow
workflow Test-Workflow {
$obj = New-Object -TypeName PSObject
$obj | Add-Member -MemberType NoteProperty `
-Name ExampleProperty `
-Value 'Hello!'
$obj | Get-Member
}
Test-Workflow
eg3 : workflow using InlineScript
Import-Module PSWorkflow
workflow Test-Workflow {
InlineScript {
$obj = New-Object -TypeName PSObject
$obj | Add-Member -MemberType NoteProperty `
-Name ExampleProperty `
-Value 'Hello!'
$obj | Get-Member
}
}
Test-Workflow
```

18.1.3 Persisting State

Checkpoint-Workflow

or

Persist workflow activity

or

with the -PSPersist switch

18.1.4 Suspending and resuming workflows

Suspend-Workflow (within the workflow)

Resume-Job (providing the necessary job ID)

18.1.5 Inherently remotable

可以远程运行workflow, 但是以下命令只会在本地运行 ：

- Add-Member
- Compare-Object
- ConvertFrom-Csv, ConvertFtom-Json, ConvertFrom-StringData
- Convert-Path
- ConvertTo-Csv, ConvertTo-Html, ConvertTo-Xml
- ForEach-Object
- Get-Host
- Get-Member
- Get-Random
- Get-Unique
- Group-Object
- Measure-Command
- Measure-Object
- New-PSSessionOption, New-PSTransportOption
- New-TimeSpan
- Out-Default, Out-Host, Out-Null, Out-String
- Select-Object
- Sort-Object
- Update-List
- Where-Object
- Write-Debug, Write-Error, Write-Host, Write-Output, Write-Progress, Write-Verbose, Write-Warning

如果需要在远程运行，要包在 InlineScript{} 里面。

18.1.6 Parallellism

Parallel{} 里面的语句执行顺序是随机的，如果想按顺序来，再包一层 Sequence{}。

而 parallelized ForEach 则有点不同

eg ：

```
Workflow Test-Workflow {
```

```
Foreach  -parallel ($computer in $computerName) {
Do-Something  -computerName $computer
    }
}
```

远程端个电脑执行的顺序是随机的

18.2 General workflow design strategy

18.3 Example workflow scenario

18.4 Writing the workflow

eg :

```
workflow Set-LOBAppConfiguration {
parallel {
    InlineScript {
        New-Item -Path HKLM:\SOFTWARE\Company\LOBApp\Settings
        New-ItemProperty -Path HKLM:\SOFTWARE\Company\LOBApp\Settings `
        -Name Rebuild `
        -Value 0
    }
    InlineScript {
        Set-Service -Name LOBApp -StartupType Automatic
        Start-Service -Name LOBApp
    }
    InlineScript {
        Register-PSSessionConfiguration `
            -Path C:\CorpApps\LOBApp\LOBApp.psc1 `
            -Name LOBApp
        }
        InlineScript {
            Import-Module LOBAppTools
            Set-LOBRebuildMode -Mode 1
        }
    }
}
```

eg :

```
Set -LOBAppConfiguration -PSComputerName one,two,three
```

18.5 Workflows vs. functions

1. workflows 和 functions 的不同之处

1）不能用 begin, process, end

2）不能用子表达式，如 $myvar = "$($service.name)"

3）不能用 drive-qualified 变量，如 $env:computername; use Get-Content ENV:ComputerName instead.

4）变量名只能包含字母，数字，下划线，连字符

5）不能执行对象的方法 （可以包在 InlineScript块里执行）

6）不能分配值给对象属性

7）You can’t dot source scripts or use the invocation (&) operator.

8）内层 workflow (被包在另一个 workflow里的workflow)不支持高级函数的参数验证（如必须属性等），但最外层的 workflow 是支持这些特性的。

9）不支持自动对应参数值（给出参数值是，不能省略参数名，如Dir C:\ won’t work, but Dir -Path C:\ will.）

10）不支持错误处理的Trap, 用 Try... Catch... Finally 来代替。

11）Switch 语句和函数里功能不同

12）workflow 不支持基于 comment 的帮助。

13）修改变量需要指定对应的 workflow 名，如 $workflow:myvar

2. workflow 里不能使用的本地 PowerShell 命令 ：

- Get-Alias, Export-Alias, Import-Alias, New-Alias, Set-Alias
- Update-FormatData
- Add-History, Clear-History, Get-History, Invoke-History
- New-PSDrive, Remove-PSDrive
- Set-StrictMode
- Start-Transcript, Stop-Transcript
- Remove-TypeData, Update-TypeData
- Clear-Variable, Get-Variable, New-Variable

18.6 Lab

19 Troubleshooting pipeline input

19.1 Refresher: how pipeline input works

help stop-service -full : get help of the stop-service command.

19.2 Introducing Trace-Command

to capture and display some internals about what it's doing.

format :

Trace-Command -Name Parameterbinding -PSHost -Expression { }

note : expression 块里面的命令是会被执行的。

19.3 Interpreting trace-command output

eg :

Trace-Command -name ParameterBinding -PSHost -Expression { Import -Csv .\computers.csv | Get-Service -Name * }

19.4 Lab

20. Using object hierarchies for complex output

20.1 When a hierarchy might be necessary

20.2 Hierarchies and CSV: not a good idea

eg :

Get-Service | Export-CSV services.csv

This is what happens when PowerShell has to convert a hierarchy of objects into a flatfile format like CSV.

20.3 Creating nested objects

20.4 Working with nested objects

four main techniques :

1) use Select-Object to expand a property that contains busobjects, enabling you to see the individual subobjects.

2) use Format-Custom to expand the entire object hierarchy.

3) use a ForEach loop.

4) use PowerShell's array syntax to work with infividual subobjects.

20.4.1 Using Select-Object to expand child objects

eg :

Get-Service | select -ExpandProperty ServiceDependedOn

eg :

Get-service -Name BITS | select -ExpandProperty ServiceDependedOn

20.4.2 Using Format-Custom to expand an object hierarchy

eg : to expand each service

get-service | format-custom -Property *

20.4.3 Using a ForEach loop to enumarate subobjects

eg :

```
$services = Get-Service
foreach ($main_service in $services) {
    Write " $($main_service.name) depends on:"
    foreach ($sub_service in $main_service.requiredservices) {
        Write "`t $($sub_service.name)"
    }
}
```

20.4.4 Using PowerShell's array syntax to access individual subobjects

eg : pull a list of all services into $services. then access the fifth service (index number 4), its RequiredServices property, the first required service (index number 0), and that service's Name proeprty.

```
$services = get-service
$services[4].requiredservice[0].name
```

20.5 Lab

21 Globalizing a function

21.1 Introduction to globalization

1) a data section

2) two built-in variables

$PSCulture : contains the language used for regional settings such as date, time and currency formats

$PSUICulture : contains the language for user interface elements such as menus and text strings.

3) ConvertForm-StringData : a cmdlet that converts text strings into a hash table

4) The .psd1 file type

5) Import-LocalizedData : a cmdlet that imports translated text strings for a specific language into a script.

eg : a starting point, Global.psm1

```
function Get-OSInfo {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
        [string[]]$computerName
    )
    BEGIN {
        Write-Verbose "Starting Get-OSInfo"
    }
    PROCESS {
        ForEach ($computer in $computername) {
            try {
                $connected = $True
                Write-Verbose "Attempting $computer"
                $os = Get-WmiObject -ComputerName $computer `
                                    -class Win32_OperatingSystem `
                                    -EA Stop
            } catch {
                $connected = $false
                Write-Verbose "Connection to $computer failed"
            }
        if ($connected) {
            Write-Verbose "Connection to $computer succeeded"
            $cs = Get-WmiObject -ComputerName $computer `
                                -class Win32_ComputerSystem
            $props = @{ComputerName=$computer;
                       OSVersion=$os.version;
                       Manufacturer=$cs.manufacturer;
                       Model=$cs.model}
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
}
END {
```

```
    Write-Verbose "Ending Get-OSInfo"
    }
}
```

21.2 PowerShell's data language

data section

ConvertFrom-StringData : a cmdlet to convert a here-string into a hashtable.

21.3 Storing translated strings

eg : de-DE version of Global.psd1

```
    ConvertFrom-StringData @'
        attempting = Versuch
        connectionTo = Der anschluss an
        failed = gescheitert
        succeeded = gelungen
        starting = Ab Get-OSInfo
        ending = Ende Get-OSInfo
'@
```

eg : es version of Global.psd1

```
    ConvertFrom-StringData @'
        attempting = Intentar
        connectionTo = Conexion a
        failed = fracasado
        succeeded = exito
        starting = A partir Get-OSInfo
        ending = Final Get-OSInfo
'@
```

NOTE : The closing '@ can't be indented.

Import-LocalizedData : a cmdlet to load the translated data.

eg :

```
Import-LocalizedData -BindingVariable $msgTable
```

eg :

```
Export-ModuleMember -function "Get-OSInfo"
```

21.4 Do you need to globalize?

21.5 Lab

22 Crossing the line: utilizing the .NET Framework

22.1 .NET classes and instances

22.2 Static methods of a class

A static method is one that's accessible as part of the class itself, without actually creating an instance of the class.

eg :

```
[System.Math]::Abs(-5)
```

or

```
[math]::abs(-5)
```

note :

Most of the classes under the top-level System namespace are available by default, but for other namespaces you may need to explictly load the necessary assembly in order to begin using the classes.

eg :

```
[microsoft.visualbasic.vbmath]::rnd()
```

eg : to manually load the necessary assembly

```
[system.reflection.assembly]::loadwithpartialname['Microsoft.Visual Basic') | Out-Null
```

22.3 Instantiating a class

New-Object : a cmdlet to create a new instance of a class.

eg :
$drive = New-Object -TypeName System.IO.DriveInfo -Argument 'C:'

22.4 Using Reflection
Get-Member : a cmdlet to utilize a .NET Framework feature called Reflection.
Reflection lets you see an object's members -- its properties, methods, and events -- simply by looking at it. (This applies only to instances.)
eg : create an instance of the System.IO.DriveInfo class, pointing it to our C:drive, and then ask Get-Member to show us the instance's members
$drive = New-Object -TypeName System.IO.DriveInfo -ArgumentList 'C:'
$drive | Get-Member

22.5 Finding class documentation
Google, Bing, or some other search engine is your best bet for finding .NET Framework documentation. You can also start at http://msdn.microsoft.com/en-us/ library/gg145045, which is the top-level page (at the time of this writing) for the entire Framework's documentation library.
Note that there are more than a few versions of the .NET Framework out there; PowerShell v3 uses Framework v4, so you'll often want to look specifically at documentation for version 4.

22.6 PowerShell vs. Visual Studio

22.7 Lab
The .NET Framework contains a class named Dns, which lives within the System.Net namespace. Read its documentation at http://msdn.microsoft.com/en-us/library/ system.net.dns. Pay special attention to the static GetHostEntry() method. Use this method to return the IP address of www.MoreLunches.com.

Part 4 Creating tools for delegated administration

23 Creating a GUI too, part 1: the GUI
Using Windows Forms (WinForms), one of two .NET Framework GUI systems.
on a commercial tool named PowerShell Studio.

23.1 Introduction to WinForms
We suggest Microsoft's own MSDN Library as a great reference to WinForms; visit http://msdn.microsoft.com/en-us/library/cc656767 for the WinForms Portal.

23.2 Using a GUI to create the GUI
PowerShell Studio

23.3 Manually coding the GUI
see the script created using PowerShell Studio's Export to Clipboard File option
1) the section that loads the .NET Framework pieces required to create the GUI.
2) the code to create the various GUI elements we designed.
3) several sections that assign our initial property values.

23.4 Showing the GUI

23.5 Lab

24 Creating a GUI tool, part 2: the code
Adding the code needed to make that GUI functional.

24.1 Addressing GUI objects

24.2 Example: text boxes
eg : $ComputerName.Text
eg : Change event

eg : to make computer name text box default to the local computer

```
$formMenu_Load = {
    #TODO: Initialize Form Controls here
    $ComputerName.txt=$env:computername
}
```

24.3 Example: button clicks

eg :

```
$OKButton_Click={
    if ($EventLogName.Visible) {
        # retrieve event log
    } else {
        # populate event log list
        $logs = Get-EventLog -ComputerName $ComputerName.Text `
                        -List
    }
}
```

24.4 Example: list boxes

eg :

```
$OKButton_Click={
    if ($EventLogName.Visible) {
        # retrieve event log
        if ($EventLogName.SelectedIndex -gt -1) {
            $entries = Get-EventLog -Computer $ComputerName.Text `
                                -Log $EventLogName.SelectedItem
        }
    } else {
        # populate event log list
        $logs = Get-EventLog -ComputerName $ComputerName.Text `
                            -List | Select-Object -ExpandProperty Log
        Load-ComboBox -ComboBox $EventLogName `
                        -Items $logs
        $EventLogName.Visible = $true
        $labelSelectEventLog.Visible = $true
    }
}
```

24.5 Example: radio buttons

eg :

```
$buttonOK_Click={
    #TODO: Place custom script here
    if ($radiobuttonServices.Checked) {
        $Title="Services for $($computername.Text)"
        Get-Service -ComputerName $Computername.Text | Out-GridView -Title
➥$Title
    }
    elseif ($radiobuttonProcesses.Checked) {
        $Title="Processes for $($computername.Text)"
        Get-Process -ComputerName $Computername.Text | Out-GridView -Title
➥$Title
    }
    elseif ($radiobuttonDiskSpace.Checked) {
        $Title="DiskSpace for $($computername.Text)"
        Get-WMIObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName $Computername.Text |
        Select DeviceID,Size,Freespace,Volume Out-GridView -Title $Title
    }
    else {
        #this should never happen
        Write-Warning "Failed to determine what radio button is checked"
    }
}
```

24.6 Example: check boxes

eg :

```
$buttonOK_Click={
    #TODO: Place custom script here
    if ($checkboxPing.Checked) {
        if (Test-Connection -ComputerName $Computername.Text -Quiet) {
            $pinged=$true
        }
        else {
            Write-Warning "Failed to ping $($computername.text)"
            $pinged=$False
        }
    }
    else {
        #don't test ping, assume we can.
        $pinged=$True
    }
    If ($pinged) {
        $data=Get-WmiObject -Class Win32_ComputerSystem -ComputerName
        $Computername.Text
    }
    if ($checkboxLogResults.Checked) {
        $export=Join-Path -Path "$env:userprofile\documents" -ChildPath
➥"$($Computername.Text)-CS.xml"
        $data | Export-Clixml -Path $export
        Get-Item $export | out-gridview
    }
    else {
        $data | out-gridview
    }
}
```

24.7 Lab

First, set the Computername text box so that it defaults to the actual local computer name. Don't use localhost. Then, connect the OK button so that it runs the Get-ServiceData function from the lab in chapter 23 and pipes the results to the pipeline. You can modify the function if you want. Use the form controls to pass parameters to the function.

25. Creating a GUI tool, part 3: the output

25.1 Using Out-GridView

Out-GridView : a cmdlet which will be available on any computer that has the PowerShell ISE installed.

25.5 Creating a form for output

eg:

```
$OKButton_Click={
    if ($EventLogName.Visible) {
        # retrieve event log
        if ($EventLogName.SelectedIndex -gt -1) {
            $entries = Get-EventLog -Computer $ComputerName.Text `
                                    -Log $EventLogName.SelectedItem
            # $entries | Out-GridView
        }
    } else {
        # populate event log list
        $logs = Get-EventLog -ComputerName $ComputerName.Text `
                             -List | Select-Object -ExpandProperty Log
        Load-ComboBox -ComboBox $EventLogName `
                             -Items $logs
        $EventLogName.Visible = $true
        $labelSelectEventLog.Visible = $true
    }
}
```

25.3 Populating and showing the output

the Main() function is what gets the script up and running and displays the initial form window. That's handled by the Call-MainForm_pff function
Call -Results_pff

25.4 Lab

26 Creating proxy functions

26.1 What are proxy functions?
A proxy function or a wrapper function is designed to sit on top of an existing command.

26.2 Creating the proxy function template

26.3 Removing a parameter

26.4 Adding a parameter

26.5 Loading the proxy function

26.6 Lab

27 Setting up constrained remoting endpoints

27.1 Refresher: Remoting architecture

27.2 What are constrained endpoints?
A constrained endpoint is an endpoint that has limited capabilities.

27.3 Creating the endpoint definition

27.4 Registering the endpoint
eg :
Register-PSSessionConfiguration -Path C:\NetTechEndpoint.pssc -Name NetTechs -ShowSecurityDescriptorUI -AccessMode Remote -RunAsCredential Administrator
eg :
Get-PSSessionConfiguration -Name Net*

27.5 Connecting to the endpoint
eg:
Enter-PSSession -ComputerName localhost -ConfigurationName nettechs

27.6 Lab

28 Never the end

28.1 Welcomet to toolmaking

28.2 Cool ideas for tools

28.3 What's your next step?