Labs

Learn PowerShell Toolmaking
in a Month of Lunches

Chapter 4

Lab

Simple Scripts and Functions

Lab Time: 20

Lab A

WMI is a great management tool and one we think toolmakers often take advantage of. 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, e.g. Running. The whole point of tool-making is to remain flexible and re-usable without having to constantly edit the script . You should already have a start based on the examples in this chapter.

Lab B

For your second lab, look at this script:

```
Function Get-DiskInfo {
Param ([string]$computername='localhost',[int]$MinimumFreePercent=10)
$disks=Get-WmiObject -Class Win32_Logicaldisk -Filter "Drivetype=3"
foreach ($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
```

Pretty hard to read and follow, isn't it? Grab the file from the MoreLunches site, open it in the ISE and reformat it to make it easier to read. Don't forget to verify it works.

Chapter 5

Lab

Scope

Lab Time: 15

This script is supposed to create some new PSDrivesbased on environmental variables like %APPDATA% and%USERPROFILE%\DOCUMENTS. However, after the script runs the drives don't exist. Why? What changes would you make?

```
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
```

Chapter 6

Lab

Tool Design Guidelines

Lab Time: 30

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on More-Lunches.com before proceeding to the next lab in the sequence.

In this lab, we aren't going to have you write any actual scripts or functions. Instead, we want you to think about the design aspect, something many people overlook. Let's say you've been asked to develop the following PowerShell tools. Even though the tool will be running from PowerShell 3.0, you don't have to assume that any remote computer is running PowerShell 3.0. Assume at least PowerShell v2.

Lab A

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 file name 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 (for AdminPasswordStatus in Win32_ComputerSystem)
- Model
- Manufacturer
- BIOSSerial (for SerialNumber in Win32_BIOS)
- OSVersion (for Version in Win32_OperatingSystem)
- SPVersion (for ServicePackMajorVersion in Win32_OperatingSystem)

Again, you aren't writing the script only outlining what you might do..

Lab B

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 2 decimal places). Only include volumes that represent fixed hard drives – do not include optical or network drives in the output. Keep in mind that any given 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 file name but defaulting to C:\Errors.txt. Also, plan to create a custom view in the future so that your function always outputs a table, using the following column headers:

- ComputerName
- Drive
- FreeSpace
- Size

Lab C

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. However, peak page file usage and thread count will not display by default.

For each tool, think about the following design questions:

- What would be a good name for your tool.
- What sort of information do you need for each tool? (These might be potential parameters)
- How do you think the tool would be run from a command prompt or what type of data will it write to the pipeline??

Chapter 7

Lab

Advanced Functions, Part 1

Lab Time: 60

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on More-Lunches.com before proceeding to the next lab in the sequence.

Using your design notes from the previous chapter, start building your tools. You won't have to address every single design point right now. We'll revise and expand these functions a bit more in the next few chapters. For this chapter your functions should complete without error, even if they are only using temporary output.

Lab A

Using your notes from Lab A in 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. For now, keep each property's name, using ServicePackMajorVersion, Version, SerialNumber, etc. But go ahead and "translate" the value for AdminPasswordStatus to the appropriate text equivalent.

Test the function by adding <function-name> -computerName localhost to the bottom of your script, and then running the script. The output for a single computer should look something like this:

```
Workgroup                 :
Manufacturer              : innotek GmbH
Computername              : CLIENT2
Version                   : 6.1.7601
Model                     : VirtualBox
AdminPassword             : NA
ServicePackMajorVersion   : 1
BIOSSerial                : 0
```

It is possible that some values may be empty.

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 2 decimal points. Test the function by adding <function-name> -computerName localhost to the bottom of your script, and then running the script. The output for a single service should look something like this:

```
FreeSpace    Drive                    Computername            Size
---------    -----                    ------------            ----
0.07         \\?\Volume{8130d5f3...   CLIENT2                 0.10
9.78         C:\Temp\                 CLIENT2                 10.00
2.72         C:\                      CLIENT2                 19.90
2.72         D:\                      CLIENT2                 4.00
```

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, and 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.

The output for a single service should look something like this:

```
Computername       : CLIENT2
ThreadCount        : 52
ProcessName        : svchost.exe
Name               : wuauserv
VMSize             : 499138560
PeakPageFile       : 247680
Displayname        : Windows Update
```

Standalone Lab

If time is limited, you can skip the 3 labs above and work on this single, stand-alone lab. Write an advanced function named Get-SystemInfo. This function should accept one or more computer names 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:** The last boot time property does not contain a human-readable date/time value; you will need to use the class' 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:

```
Model              : VirtualBox
ComputerName       : localhost
Manufacturer       : innotek GmbH
LastBootTime       : 6/19/2012 8:55:34 AM
OSVersion          : 6.1.7601
```

Chapter 8

Lab

Advanced Functions, Part 2

Lab Time: 60

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on More-Lunches.com before proceeding to the next lab in the sequence.

In this chapter we're going to build on the functions you created in the last chapter using the concepts you hopefully picked up today. As you work through these labs, add verbose messages to display key steps or progress information.

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 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:

```
VERBOSE: Starting Get-Computerdata
VERBOSE: Getting data from localhost
VERBOSE: Win32_Computersystem
VERBOSE: Win32_Bios
VERBOSE: Win32_OperatingSystem

Workgroup                :
Manufacturer             : innotek GmbH
Computername             : CLIENT2
Version                  : 6.1.7601
Model                    : VirtualBox
AdminPassword            : NA
ServicePackMajorVersion : 1
BIOSSerial               : 0

VERBOSE: Ending Get-Computerdata
```

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:

```
VERBOSE: Starting Get-VolumeInfo
VERBOSE: Getting volume data from localhost
VERBOSE: Procssing volume \\?\Volume{8130d5f3-8e9b-11de-b460-806e6f6e6963}\

FreeSpace                      Drive                    Computername
    Size
---------                      -----                    ---------
    ---
0.07                           \\?\Volume{8130d5f3...   [redacted]
    0.10
VERBOSE: Procssing volume C:\Temp\
9.78                           C:\Temp\                 [redacted]
    10.00
VERBOSE: Procssing volume C:\
2.72                           C:\                      2TNEILC
    19.90
VERBOSE: Procssing volume D:\
2.72                           D:\                      2TNEILC
    4.00
VERBOSE: Ending Get-VolumeInfo
```

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:

```
VERBOSE: Starting Get-ServiceInfo
VERBOSE: Getting services from localhost
VERBOSE: Processing service AudioEndpointBuilder

Computername       : CLIENT2
ThreadCount        : 13
ProcessName        : svchost.exe
Name               : AudioEndpointBuilder
VMSize             : 172224512
PeakPageFile       : 83112
Displayname        : Windows Audio Endpoint Builder
```

Standalone Lab

Use this script as your starting point:

```
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.Last-
   BootupTime));
                       '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 for should look something like this:

```
VERBOSE      : Getting WMI data from localhost

Model              : VirtualBox
ComputerName       : localhost
Manufacturer       : innotek GmbH
LastBootTime       : 6/19/2012 8:55:34 AM
OSVersion          : 6.1.760
```

Chapter 9

Lab

Writing Help

Lab Time: 20

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on More-Lunches.com before proceeding to the next lab in the sequence.

These labs will build on what you've already created, applying new concepts from this chapter.

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.

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 –ComputerName parameter. Test your help by adding help <function-name> to the end of your script.

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.

Standalone Lab

Using the script in Listing 9.2 add comment-based help.

List 9.2 Standalone lab starting point

```
function Get-SystemInfo {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName
    )
    PROCESS {
        foreach ($computer in $computerName) {
            Write-Verbose “Getting WMI data from $computer”
            $os = Get-WmiObject -class Win32_OperatingSystem -computerName
   $computer
            $cs = Get-WmiObject -class Win32_ComputerSystem -computerName $com-
   puter
            $props = @{‘ComputerName’=$computer
                       ‘LastBootTime’=($os.ConvertToDateTime($os.Last-
   BootupTime))
                       ‘OSVersion’=$os.version
                       ‘Manufacturer’=$cs.manufacturer
                       ‘Model’=$cs.model
                   }
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
}

Include at least a synopsis, description, and help for the –ComputerName pa-
   rameter. Test your help by adding help <function-name> to the end of your
   script.
```

Chapter
10

Lab

Error Handling

Lab Time: 60

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on MoreLunches.com before proceeding to the next lab in the sequence.

You are going to continue with the functions you've been building the last few chapters. The next step is to begin incorporating some error handling using Try/Catch/Finally. If you haven't done so, take a few minutes to read the help content on Try/Catch/Finally. For any changes you make, don't forget to update your comment-based help.

Lab A

Using Lab A from Chapter 9, add a –ErrorLog parameter to your advanced function, which accepts a filename for an error log and defaults to C:\Errors.txt. When the function is run with this parameter, failed computer names should be appended to the error log file.

Next, if the first WMI query fails, the function should output nothing for that computer and should not attempt a second or third WMI query. Write an error to the pipeline containing each failed computer name.

Test all of this by adding this line <function-name> -ComputerName localhost,NOTONLINE –verbose to the end of your script. A portion of the output should look something like this:

```
VERBOSE: Starting Get-Computerdata
VERBOSE: Getting data from localhost
VERBOSE: Win32_Computersystem
VERBOSE: Win32_Bios
VERBOSE: Win32_OperatingSystem

Workgroup                 :
Manufacturer              : innotek GmbH
Computername              : CLIENT2
Version                         : 6.1.7601
SerialNumber              : 0
Model                           : VirtualBox
AdminPassword             : NA
ServicePackMajorVersion   : 1
```

```
VERBOSE: Getting data from notonline
VERBOSE: Win32_Computersystem
Get-Computerdata : Failed getting system information from notonline. The RPC
   server is
unavailable. (Exception from HRESULT: 0x800706BA)
At S:\Toolmaking\Ch10-LabA.ps1:115 char:40
+ 'localhost','notonline','localhost' |  Get-Computerdata -logerrors -verbose
+                                        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.
   WriteErrorException,Get-Comp
   uterData

VERBOSE: Getting data from localhost
```

Lab B

Using Lab B from Chapter 9, add a –ErrorLog parameter to your advanced function, which accepts a filename for an error log and defaults to C:\Errors.txt. When the function is run with this parameter, failed computer names should be appended to the error log file.

Test all of this by adding this line <function-name> -ComputerName localhost,NOTONLINE –verbose to the end of your script. A portion of the output should look something like this:

```
VERBOSE: Starting Get-VolumeInfo
VERBOSE: Getting data from localhost

FreeSpace       Drive                   Computername    Size
---------       -----                   ------------    ----
0.07            \\?\Volume{8130d5f3...  CLIENT2         0.10
9.78            C:\Temp\                CLIENT2         10.00
2.72            C:\                     CLIENT2         19.90
2.72            D:\                     CLIENT2         4.00

VERBOSE: Getting data from NotOnline
Get-VolumeInfo : Failed to get volume information from NotOnline. The RPC server
   is unavailable. (Exception from HRESULT: 0x800706BA)
At S:\Toolmaking\Ch10-LabB.ps1:96 char:27
+ 'localhost','NotOnline' | Get-VolumeInfo -Verbose -logerrors
+                           ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.
   WriteErrorException,Get-VolumeInfo

VERBOSE: Logging errors to C:\Errors.txt
VERBOSE: Ending Get-VolumeInfo
```

Lab C

Using Lab C from Chapter 9, add a –LogErrors switch parameter to your advanced function. Also add a –ErrorFile parameter, which accepts a filename for an error log and defaults to C:\Errors.txt. When the function is run with the -LogErrors parameter, failed computer names should be appended to the error log file. .Also, if –LogErrors is used, the log file should be deleted at the start of the function if it exists, so that each time the command starts with a fresh log file.

Test all of this by adding this line <function-name> -ComputerName localhost,NOTONLINE –verbose –logerrors to the end of your script. A portion of the output should look something like this:

```
VERBOSE: Processing service wuauserv
VERBOSE: Getting process for wuauserv
Computername : CLIENT2
ThreadCount  : 45
ProcessName  : svchost.exe
Name         : wuauserv
VMSize       : 499363840
PeakPageFile : 247680
Displayname  : Windows Update

VERBOSE: Getting services from NOTOnline
Get-ServiceInfo : Failed to get service data from NOTOnline. The RPC server is
unavailable. (Exception from HRESULT: 0x800706BA)
At S:\Toolmaking\Ch10-LabC.ps1:109 char:39
+ “localhost”,”NOTOnline”,”localhost” | Get-ServiceInfo -logerrors -verbose
+                                       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.
   WriteErrorException,Get-ServiceInfo

VERBOSE: Logging errors to C:\Errors.txt
VERBOSE: Getting services from localhost
VERBOSE: Processing service AudioEndpointBuilder
VERBOSE: Getting process for AudioEndpointBuilder
```

Standalone Lab

Use the code in Listing 10.4 as a starting point.

Listing 10.4 Standalone lab starting point

```
Function Get-SystemInfo {

<#
.SYNOPSIS
Gets critical system info from one or more computers.
.DESCRIPTION
This command uses WMI, and can accept computer names, CNAME aliases, and IP ad-
   dresses. WMI must be enabled and you must run this with admin rights for any
   remote computer.
.PARAMETER Computername
One or more names or IP addresses to query.
.EXAMPLE
Get-SystemInfo -computername localhost
#>
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName
    )
    PROCESS {
        foreach ($computer in $computerName) {
            WWrite-Verbose “Getting WMI data from $computer”
            $os = Get-WmiObject -class Win32_OperatingSystem -computerName
   $computer
            $cs = Get-WmiObject -class Win32_ComputerSystem -computerName $com-
   puter
            $props = @{‘ComputerName’=$computer
                       ‘LastBootTime’=($os.ConvertToDateTime($os.Last-
   BootupTime))
                       ‘OSVersion’=$os.version
                       ‘Manufacturer’=$cs.manufacturer
                       ‘Model’=$cs.model
                      }
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
}
```

Add a –LogErrors switch to this advanced function. When the function is run with this switch, failed computer names should be logged to C:\Errors.txt. This file should be deleted at the start of the function each time it is run, so that it starts out fresh each time. If the first WMI query fails, the function should output nothing for that computer and should not attempt a second WMI query. Write an error to the pipeline containing each failed computer name.

Test your script by adding this line to the end of your script.

```
Get-SystemInfo -computername localhost,NOTONLINE,localhost -logerrors
A portion of the output should look something like this:
Model        : VirtualBox
ComputerName : localhost
Manufacturer : innotek GmbH
LastBootTime : 6/19/2012 8:55:34 AM
OSVersion    : 6.1.7601

Get-SystemInfo : NOTONLINE failed
At S:\Toolmaking\Ch10-Standalone.ps1:51 char:1
+ Get-SystemInfo -computername localhost,NOTONLINE,localhost -logerrors
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorExcep-
   tion
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.
   WriteErrorException,Get-Syst
   emInfo

Model        : VirtualBox
ComputerName : localhost
Manufacturer : innotek GmbH
LastBootTime : 6/19/2012 8:55:34 AM
OSVersion    : 6.1.7601
```

Chapter 11

Lab

Debugging Techniques

Lab Time: 45

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

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

The function in Listing 11.8 is supposed to display some properties of running services sorted by the service account.

The function in listing 11.9 is a bit more involved. It's designed to get recent event log entries for a specified log on a specified computer. Events are sorted by the event source and added to a log file. The filename is based on the date, computer name, and event source. At the end, the function displays a directory listing of the logs. Hint: Clean up the formatting first.

Lab A

Listing 11.8 A broken function

```
1  Function Get-ServiceInfo {
2  [cmdletbinding()]
3  Param([string]$Computername)
4  $services=Get-WmiObject -Class Win32_Services -filter “state=’Running” `
    -computername $computernam
5  Write-Host “Found ($services.count) on $computername” –Foreground Green
6  $sevices | sort -Property startname,name  Select -property `
   startname,name,startmode,computername
7  }
```

Lab B

Listing 11.9 Buggy Export Function

```
01  Function Export-EventLogSource {
02
03  [cmdletbinding()]
04  Param (
05  [Parameter(Position=0,Mandatory=$True,Helpmessage="Enter a computername",Va
    lueFromPipeline=$True)]
06  [string]$Computername,
07  [Parameter(Position=1,Mandatory=$True,Helpmessage="Enter a classic event
    log name like System")]
08  [string]$Log,
09  [int]$Newest=100
10  )
11  Begin {
12  Write-Verbose "Starting export event source function"
13  #the date format is case-sensitive"
14  $datestring=Get-Date -Format "yyyyMMdd"
15  $logpath=Join-path -Path "C:\Work" -ChildPath $datestring
16  if (! (Test-Path -path $logpath) {
17  Write-Verbose "Creating $logpath"
18  mkdir $logpath
19  }
20  Write-Verbose "Logging results to $logpath"
21  }
22  Process {
23  Write-Verbose "Getting newest $newest $log event log entries from $computername"
24  Try {
25  Write-Host $computername.ToUpper -ForegroundColor Green
26  $logs=Get-EventLog -LogName $log -Newest $Newest -Computer $Computer -Er-
    rorAction Stop
27  if ($logs) {
28  Write-Verbose "Sorting $($logs.count) entries"
29  $log | sort Source | foreach {
30  $logfile=Join-Path -Path $logpath -ChildPath "$computername-$($_.Source).txt"
31  $_ | Format-List TimeWritten,MachineName,EventID,EntryType,Message |
32  Out-File -FilePath $logfile -append
33
34  #clear variables for next time
35  Remove-Variable -Name logs,logfile
36  }
37  else {Write-Warning "No logged events found for $log on $Computername"}
38  }
39   Catch { Write-Warning $_.Exception.Message }
40  }
```

```
41  End {dir $logpath
42  Write-Verbose “Finished export event source function”
43  }
44  }
```

Chapter 12

Lab

Creating Custom Format Views

Lab Time: 60

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on More-Lunches.com before proceeding to the next lab in the sequence.

We bet you can guess what is coming. You'll be adding type information and creating custom format files for the functions you've been working on the last several chapters. Use the dotnettypes.format.ps1xml and other .ps1xml files as sources for sample layout. Copy and paste the XML into your new format file. Don't forget that tags are case-sensitive.

Lab A

Modify your advanced function from Lab A in Chapter 10 so that the output object has the type name MOL.ComputerSystemInfo. Then, create a custom view in a file named C:\CustomViewA.format.ps1xml. The custom view should display objects of the type MOL.ComputerSystemInfo in a list format, displaying the information in a list as indicated in your design for this lab. Go back to Chapter 6 to check what the output names should be.

At the bottom of the script file, add these commands to test:

```
Update-FormatData –prepend c:\CustomViewA.format.ps1xml
<function-name> -ComputerName localhost
```

The final output should look something like the following.

```
Computername     : CLIENT2
Workgroup        :
AdminPassword    : NA
Model            : VirtualBox
Manufacturer     : innotek GmbH
BIOSSerialNumber : 0
OSVersion        : 6.1.7601
SPVersion        : 1
```

Note that the list labels are not exactly the same as the custom object's property names.

Lab B

Modify your advanced function Lab B from Chapter 10 so that the output object has the type name MOL.DiskInfo. Then, create a custom view in a file named C:\CustomViewB.format.ps1xml. The custom view should display objects of the type MOL.DiskInfo in a table format, displaying the information in a table as indicated in your design for this lab. Refer back to Chapter 6 for a refresher. The column headers for the FreeSpace and Size properties should display "FreeSpace(GB)" and "Size(GB)," respectively.

At the bottom of the script file, add these commands to test:

```
Update-FormatData –prepend c:\CustomViewB.format.ps1xml
<function-name> -ComputerName localhost
```

The final output should look something like the following.

```
ComputerName       Drive                          FreeSpace(GB)   Size(GB)
------------       -----                          -------------   --------
CLIENT2            \\?\Volume{8130d5f3-8e9b-...   0.07            0.10
CLIENT2            C:\Temp\                       9.78            10.00
CLIENT2            C:\                            2.72            19.90
CLIENT2            D:\                            2.72            4.00
```

Note that the column headers are not exactly the same as the custom object's property names.

Lab C

Modify your advanced function Lab C from Chapter 10 so that the output object has the type name MOL.ServiceProcessInfo. Then, create a custom view in a file named C:\CustomViewC.format.ps1xml. The custom view should display objects of the type MOL.ServiceProcessInfo in a table format, displaying computername, service name, display name, process name, and process virtual size.

In addition to the table format, create a list view in the same file that displays the properties in this order:

- Computername
- Name (renamed as Service)
- Displayname
- ProcessName
- VMSize
- ThreadCount
- PeakPageFile

At the bottom of the script file, add these commands to test:

```
Update-FormatData –prepend c:\CustomViewC.format.ps1xml
<function-name> -ComputerName localhost
<function-name> -ComputerName localhost | Format-List
```

The final output should look something like this for the table.

```
ComputerName     Service          Displayname          ProcessName      VM
------------     -------          -----------          -----------      --
CLIENT2          AudioEndpo...    Windows Audio E...   svchost.exe      172208128
CLIENT2          BFE              Base Filtering ...   svchost.exe      69496832
CLIENT2          BITS             ackground Inte...    svchost.exe      499310592
CLIENT2          Browser          Computer Browser     svchost.exe      499310592
```

And like this for the list:

```
Computername : CLIENT2
Service      : AudioEndpointBuilder
Displayname  : Windows Audio Endpoint Builder
ProcessName  : svchost.exe
VMSize       : 172208128
ThreadCount  : 13
PeakPageFile : 83112
```

Note that per the design specifications from Chapter 6 not every object property is displayed by default and that some column headings are different than the actual property names.

Chapter
13

Lab

Script and Manifest Modules

Lab Time: 30

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on More-Lunches.com before proceeding to the next lab in the sequence.

In this chapter you are going to assemble a module called PSHTools, from the functions and custom views that you've been working on for the last several chapters. Create a folder in the user module directory, called PSHTools. Put all of the files you will be creating in the labs into this folder.

Lab A

Create a single ps1xml file that contains all of the view definitions from the 3 existing format files. Call the file PSHTools.format.ps1xml. You'll need to be careful. Each view is defined by the <View></View> tags. These tags, and everything in between should go between the <ViewDefinition></ViewDefinition> tags.

Lab B

Create a single module file that contains the functions from the Labs A, B and C in Chapter 12, which should be the most current version. Export all functions in the module. Be careful to copy the function only. In your module file, also define aliases for your functions and export them as well.

Lab C

Create a module manifest for the PSHTools that loads the module and custom format files. Test the module following these steps:

1. Import the module
2. Use Get-Command to view the module commands
3. Run help for each of your aliases
4. Run each command alias using localhost as the computername and verify formatting
5. Remove the module
6. Are the commands and variables gone?

Chapter 16

Lab

Making Tools that Make Changes

Lab Time: 30

In WMI, the Win32_OperatingSystem class has a method called Win32Shutdown. It accepts a single input argument, which is a number that determines if the method shuts down, powers down, reboots, and logs off the computer.

Write a function called Set-ComputerState. Have it accept one or more computer names on a –ComputerName parameter. Also provide an –Action parameter, which accepts only the values LogOff, Restart, ShutDown, or PowerOff. Finally, provide a –Force switch parameter (switch parameters do not accept a value; they're either specified or not).

When the function runs, query Win32_OperatingSystem from each specified computer. Don't worry about error handling at this point – assume each specified computer will be available. Be sure to implement support for the –WhatIf and –Confirm parameters, as outlined in this chapter. Based upon the –Action specified, execute the Win32Shutdown method with one of the following values:

- LogOff – 0
- ShutDown – 1
- Restart – 2
- PowerOff – 8

If the –Force parameter is specified, add 4 to those values. So, if the command was Set-ComputerState –computername localhost –Action LogOff –Force, then the value would be 4 (zero for LogOff, plus 4 for Force). The execution of Win32Shutdown is what should be wrapped in the implementing If block for –WhatIf and –Confirm support.

Chapter 17

Lab

Creating a Custom Type Extension

Lab Time: 30

Revisit the advanced function that you wrote for Lab A in Chapters 6 through 14 of this book. Create a custom type extension for the object output by that function. Your type extension should be a ScriptMethod named CanPing(), as outlined in this chapter. Save the type extension file as PSHTools.ps1xml. Modify the PSHTools module manifest to load PSHTools.ps1xml, and then test your revised module to make sure the CanPing() method works.

Here is a sample ps1xml file:

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
  <Type>
    <Name>MOL.ComputerSystemInfo</Name>
    <Members>
        <ScriptMethod>
        <Name>CanPing</Name>
        <Script>
        Test-Connection -ComputerName $this.ComputerName -Quiet
        </Script>
    </ScriptMethod>
    </Members>
  </Type>
</Types>
```

Chapter 19

Lab

Troubleshooting Pipeline Input

Lab Time: 20

Create a text file named C:\Computers.csv. In it, place the following content:

```
ComputerName
LOCALHOST
NOTONLINE
```

Be sure there are no extra blank lines at the end of the file. Then, consider the following command:

```
Import-CSV C:\Computers.txt | Invoke-Command –Script { Get-Service }
```

The help file for Invoke-Command indicates that its –ComputerName parameter accepts pipeline input ByValue. Therefore, our expectation is that the computer names in the CSV file will be fed to the –ComputerName parameter. But if you run the command, that isn't what happens. Troubleshoot this command using the techniques described in this chapter, and determine where the computer names from the CSV file are being bound.

Chapter 20

Lab

Using Object Hierarchies for Complex Output

Lab Time: 30

Create a new function in your existing PSHTools module. Name the new function Get-Computer-VolumeInfo. This function's output will include some information that your other functions already produce, but this particular function is going to combine them all into a single, hierarchical object.

This function should accept one or more computer names on a –ComputerName parameter. Don't worry about error handling at this time. The output of this function should be a custom object with the following properties:

- ComputerName
- OSVersion (Version from Win32_OperatingSystem)
- SPVersion (ServicePackMajorVersion from Win32_OperatingSystem)
- LocalDisks (all instances of Win32_LogicalDisk having a DriveType of 3)
- Services (all instances of Win32_Service)
- Processes (all instances of Win32_ProcessS)

The function will therefore be making at least four WMI queries to each specified computer.

Chapter
22

Lab

Crossing the Line: Utilizing the .NET Framework

Lab Time: 20

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.

Chapter 23

Lab

Creating a GUI Tool, Part 1: The GUI

Lab Time: 30

In this lab you're going to start a project that you'll work with over the next few chapters, so you'll want to make sure you have a working solution before moving on. Developing a graphical PowerShell script is always easier if you have a working command-line script. We've already done that part for you in the following listing.

LISTING CH23-LABFUNCTION

You can either retype or download the script from MoreLunches.com.

The function takes a computer name as a parameter and gets services via WMI based on user-supplied filter criteria. The function writes a subset of data to the pipeline. From the command line it might be used like this:

```
Get-servicedata $env:computername -filter running | Out-GridView
```

Your task in this lab is to create the graphical form using PowerShell Studio. You should end up with something like the form shown in figure 23.7.

Make the Running radio button checked by default. You'll find it easier later if you put the radio buttons in a GroupBox control, plus it looks cooler. The script you're creating doesn't have to do anything for this lab except display this form.

ANSWER - see code listing from MoreLunches.com chapter 23.

Chapter 24

Lab

Creating a GUI Tool, Part 2: The Code

Lab Time: 30

In this lab you're going to continue where you left off in chapter 23. If you didn't finish, please do so first or download the sample solution from MoreLunches.com. Now you need to wire up your form and put some actions behind the controls.

First, set the Computername text box so that it defaults to the actual local computer name. Don't use localhost.

TIP Look for the form's Load event function.

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.

TIP You can avoid errors if you set the default behavior to search for running services.

You can test your form by sending output to Out-String and then Write-Host. For example, in your form you could end up with a line like this:

```
<code to get data> | Out-String | Write-Host
```

In the next chapter you'll learn better ways to handle form output.

ANSWER - see code listing from MoreLunches.com chapter 24

Chapter 25

Lab

Creating a GUI Tool, Part 3: The Output

Lab Time: 20

We'll keep things pretty simple for this lab. Using the PowerShell Studio lab project from chapter 24, add a RichTextBox control to display the results. Here are some things to remember:

- Configure the control to use a fixed-width font like Consolas or Courier New.
- The Text property must be a string, so explicitly format data as strings by using-Out-String.
- Use the control's Clear() method to reset it or clear out any existing results.

If you need to move things around on your form, that's okay. You can download a sample solution at MoreLunches.com.

ANSWER - see code listing from MoreLunches.com chapter 25

Chapter
26

Lab

Creating Proxy Functions

Lab Time: 30

Create a proxy function for the Export-CSV cmdlet. Name the proxy function Export-TDF. Remove the –Delimiter parameter, and instead hardcode it to always use –Delimiter "`t" (that's a backtick, followed by the letter t, in double quotation marks).

Work with the proxy function in a script file. At the bottom of the file, after the closing } of the function, put the following to test the function:

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

Run the script to test the function, and verify that it creates a tab-delimited file named c:\services.tdf.

ANSWER - see code listing from MoreLunches.com chapter 26

Chapter
27

Lab

Setting Up Constrained Demoting Endpoints

Lab Time: 30

Create a new, local user named TestMan on your computer. Be sure to assign a pass- word to the account. Don't place the user in any user groups other than the default Users group.

Then, create a constrained endpoint on your computer. Name the endpoint ConstrainTest. Design it to include only the SmbShare module and to make only the Get-SmbShare command visible (in addition to a small core set of cmdlets like Exit-PSSession, Select-Object, and so forth). After creating the session configura- tion, register the endpoint. Configure the endpoint to permit only TestMan to con- nect (with Read and Execute permissions), and configure it to run all commands as your local Administrator account. Be sure to provide the correct password for Administrator when you're prompted.

Use Enter-PSSession to connect to the constrained endpoint. When doing so, use the –Credential parameter to specify the TestMan account, and provide the proper password when prompted. En- sure that you can run Get-SmbShare but not any other command (such as Get-SmbShareAccess).

Lab Answers

Learn PowerShell Toolmaking
in a Month of Lunches

Chapter 4

Lab Answers

Simple Scripts and Functions

Lab A

WMI is a great management tool and one we think toolmakers often take advantage of. 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, e.g. Running. The whole point of tool-making is to remain flexible and re-usable without having to constantly edit the script . You should already have a start based on the examples in this chapter.

Lab B

For your second lab, look at this script:

```
Function Get-DiskInfo {
Param ([string]$computername='localhost',[int]$MinimumFreePercent=10)
$disks=Get-WmiObject -Class Win32_Logicaldisk -Filter "Drivetype=3"
foreach ($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
```

Pretty hard to read and follow, isn't it? Grab the file from the MoreLunches site, open it in the ISE and reformat it to make it easier to read. Don't forget to verify it works.

Answers

Part 1

```
Function Get-ServiceStartMode {

Param(
[string]$Computername='localhost',
```

```
[string]$StartMode='Auto',
[string]$State='Running'
)

$filter="Startmode='$Startmode' AND state='$State'"
Get-CimInstance -ClassName Win32_Service -ComputerName $Computername -Filter
   $filter

}

#testing
Get-ServiceStartMode
Get-ServiceStartMode -Start 'Auto' -State 'Stopped'
Get-ServiceStartMode -StartMode 'Disabled' -Computername 'SERVER01'
```

Part 2

```
Function Get-DiskInfo {

Param (
[string]$computername='localhost',
[int]$MinimumFreePercent=10
)

    $disks=Get-WmiObject -Class Win32_Logicaldisk -Filter "Drivetype=3"

    foreach ($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}}

    } #close foreach

} #close function

Get-DiskInfo
```

Chapter 5

Lab Answers

Scope

This script is supposed to create some new PSDrivesbased on environmental variables like %APPDATA% and%USERPROFILE%\DOCUMENTS. However, after the script runs the drives don't exist. Why? What changes would you make?

```
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
```

Answer

The New-PSDrive cmdlet is creating the drive in the Function scope. Once the function ends the drives disappear along with the scope.The solution is to use the –Scope parameter with New-PSDrive. Using a value of Script will make them visible to the script so that the DIR command will work. However, once the script ends the drives are still removed. If the intent was to make them visible in the console, then the solution is to use a –Scope value of Global.

Chapter 6

Lab Answers

Tool Design Guidelines

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on More-Lunches.com before proceeding to the next lab in the sequence.

In this lab, we aren't going to have you write any actual scripts or functions. Instead, we want you to think about the design aspect, something many people overlook. Let's say you've been asked to develop the following PowerShell tools. Even though the tool will be running from PowerShell 3.0, you don't have to assume that any remote computer is running PowerShell 3.0. Assume at least PowerShell v2.

Lab A

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 file name 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 (for AdminPasswordStatus in Win32_ComputerSystem)
- Model
- Manufacturer
- BIOSSerial (for SerialNumber in Win32_BIOS)
- OSVersion (for Version in Win32_OperatingSystem)
- SPVersion (for ServicePackMajorVersion in Win32_OperatingSystem)

Again, you aren't writing the script only outlining what you might do..

Lab B

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 2 decimal places). Only include volumes that represent fixed hard drives – do not include optical or network drives in the output. Keep in mind that any given 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 file name but defaulting to C:\Errors.txt. Also, plan to create a custom view in the future so that your function always outputs a table, using the following column headers:

- ComputerName
- Drive
- FreeSpace
- Size

Lab C

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. However, peak page file usage and thread count will not display by default.

For each tool, think about the following design questions:

- What would be a good name for your tool.
- What sort of information do you need for each tool? (These might be potential parameters)

How do you think the tool would be run from a command prompt or what type of data will it write to the pipeline??

Answers

Lab A

Because we are getting information from a variety of WMI sources, a good function name might be Get-ComputerData. We'll need a string parameter for the name, a string for the log file and maybe a switch parameter indicating that we want to log data. The function will need to make several WMI queries and then it can write a custom object to the pipeline. We can get the computername from one of the WMI classes. We could use the computername parameter, but by using something from WMI we'll get the "official" computer name which is better if we test with something like localhost.

Since the AdminStatus property value an integer we can use a Switch statement to define a variable with the interpretation as a string.

When creating a custom object, especially one where we need to make sure property names will match the eventual custom view, a hash table will come in handy because we can use it with New-Object.

We can probably start out by having the function take computer names as parameters:

```
Get-Computerdata –computername server01,server02
```

But eventually we'll want to be able to pipe computernames to it. Each computername should produce a custom object.

Lab B

Since the command will get volume data information, a likely name would be Get-VolumeInfo or Get-VolumeData. Like Lab A we'll need a string parameter for a computername, as well as a parameter for the eventlog and a switch to indicate whether or not to log errors. A sample command might look like:

```
Get-VolumeInfo –computername Server01 –ErrorLog C:\work\errors.txt –LogError
```

Also like Lab A, using a hash table with the new properties will make it easier to create and write a custom object to the pipeline. We'll also need to convert the size and free space by dividing the size in bytes by 1GB. One way to handle the formatting requirement is to use the –f operator.

```
$Size="{0:N2}" -f ($drive.capacity/1GB)
$Freespace="{0:N2}" -f ($drive.Freespace/1GB)
```

Lab C

This lab can follow the same outline as the first two in terms of computername, error log name and whether or not to log files. Because we need to get the process id of each service, we'll need to use WMI or CIM. The Get-Service cmdlet returns a service object, but it doesn't include the process id. Once we have the service object we can execute another WMI query to get the process object.

It will most likely be easiest to create a hash table with all of the required properties from the 2 WMI classes. For now, we'll include all the properties. Later we can create a custom view with only the desired, default properties.

Since this function is getting service information, a good name might be Get-ServiceInfo.

Chapter 7

Lab Answers

Advanced Functions, Part 1

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on More-Lunches.com before proceeding to the next lab in the sequence.

Using your design notes from the previous chapter, start building your tools. You won't have to address every single design point right now. We'll revise and expand these functions a bit more in the next few chapters. For this chapter your functions should complete without error, even if they are only using temporary output.

Lab A

Using your notes from Lab A in 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. For now, keep each property's name, using ServicePackMajorVersion, Version, SerialNumber, etc. But go ahead and "translate" the value for AdminPasswordStatus to the appropriate text equivalent.

Test the function by adding <function-name> -computerName localhost to the bottom of your script, and then running the script. The output for a single computer should look something like this:

```
Workgroup                 :
Manufacturer              : innotek GmbH
Computername              : CLIENT2
Version                   : 6.1.7601
Model                     : VirtualBox
AdminPassword             : NA
ServicePackMajorVersion   : 1
BIOSSerial                : 0
```

It is possible that some values may be empty.

Using your design notes from the previous chapter, start building your tools. You won't have to address every single design point right now. We'll revise and expand these functions a bit more in the next few chapters. For this chapter your functions should complete without error, even if they are only using temporary output.

Here is a possible solution:

```
Function Get-ComputerData {

[cmdletbinding()]

param( [string[]]$ComputerName )

foreach ($computer in $computerName) {
    Write-Verbose “Getting data from $computer”
    Write-Verbose “Win32_Computersystem”
        $cs = Get-WmiObject -Class Win32_Computersystem -ComputerName $Computer

        #decode the admin password status
        Switch ($cs.AdminPasswordStatus) {
            1 { $aps=”Disabled” }
            2 { $aps=”Enabled” }
            3 { $aps=”NA” }
            4 { $aps=”Unknown” }
        }

        #Define a hashtable to be used for property names and values
        $hash=@{
            Computername=$cs.Name
            Workgroup=$cs.WorkGroup
            AdminPassword=$aps
            Model=$cs.Model
            Manufacturer=$cs.Manufacturer
        }

        Write-Verbose “Win32_Bios”
        $bios = Get-WmiObject -Class Win32_Bios -ComputerName $Computer

        $hash.Add(“SerialNumber”,$bios.SerialNumber)

        Write-Verbose “Win32_OperatingSystem”
        $os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName $Computer
        $hash.Add(“Version”,$os.Version)
        $hash.Add(“ServicePackMajorVersion”,$os.ServicePackMajorVersion)

        #create a custom object from the hash table
        New-Object -TypeName PSObject -Property $hash

} #foreach

}

Get-Computerdata -computername localhost
```

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 2 decimal points. Test the function by adding <function-name> -computerName localhost to the bottom of your script, and then running the script. The output for a single service should look something like this:

```
FreeSpace    Drive                          Computername          Size
---------    -----                          ------------          ----
0.07         \\?\Volume{8130d5f3...         CLIENT2               0.10
9.78         C:\Temp\                       CLIENT2               10.00
2.72         C:\                            CLIENT2               19.90

2.72         D:\                            CLIENT2               4.00
```

Here is a possible solution:

```
Function Get-VolumeInfo {

[cmdletbinding()]

 param( [string[]]$ComputerName )

foreach ($computer in $computerName) {

    $data = Get-WmiObject -Class Win32_Volume -computername $Computer -Filter
   “DriveType=3”

    Foreach ($drive in $data) {

        #format size and freespace in GB to 2 decimal points
        $Size=”{0:N2}” -f ($drive.capacity/1GB)
        $Freespace=”{0:N2}” -f ($drive.Freespace/1GB)

        #Define a hashtable to be used for property names and values
        $hash=@{
            Computername=$drive.SystemName
            Drive=$drive.Name
            FreeSpace=$Freespace
            Size=$Size

        }

        #create a custom object from the hash table
        New-Object -TypeName PSObject -Property $hash
    } #foreach
```

```
    #clear $data for next computer
    Remove-Variable -Name data

} #foreach computer

}

Get-VolumeInfo -ComputerName localhost
```

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, and 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.

The output for a single service should look something like this:

```
Computername       : CLIENT2
ThreadCount        : 52
ProcessName        : svchost.exe
Name               : wuauserv
VMSize             : 499138560
PeakPageFile       : 247680
Displayname        : Windows Update
```

Here is a possible solution:

```
Function Get-ServiceInfo {

[cmdletbinding()]

 param( [string[]]$ComputerName )

foreach ($computer in $computerName) {
    $data = Get-WmiObject -Class Win32_Service -computername $Computer -Filter
   "State='Running'"

    foreach ($service in $data) {

        $hash=@{
        Computername=$data[0].Systemname
        Name=$service.name
```

```
        Displayname=$service.DisplayName
        }

        #get the associated process
        $process=Get-WMIObject -class Win32_Process -computername $Computer
   -Filter "ProcessID='$($service.processid)'"
        $hash.Add("ProcessName",$process.name)
        $hash.add("VMSize",$process.VirtualSize)
        $hash.Add("PeakPageFile",$process.PeakPageFileUsage)
        $hash.add("ThreadCount",$process.Threadcount)

        #create a custom object from the hash table
        New-Object -TypeName PSObject -Property $hash

    } #foreach service

} #foreach computer

}

Get-ServiceInfo -ComputerName localhost
```

Standalone Lab

If time is limited, you can skip the 3 labs above and work on this single, stand-alone lab. Write an advanced function named Get-SystemInfo. This function should accept one or more computer names 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:** The last boot time property does not contain a human-readable date/time value; you will need to use the class' 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:

```
Model              : VirtualBox
ComputerName       : localhost
Manufacturer       : innotek GmbH
LastBootTime       : 6/19/2012 8:55:34 AM
OSVersion          : 6.1.7601
```

Here is a possible solution:

```
function Get-SystemInfo {
   [CmdletBinding()]
   param(
      [string[]]$ComputerName
   )

   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
   }
}

Get-SystemInfo -ComputerName localhost
```

Chapter

8

Lab Answers

Advanced Functions, Part 2

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on MoreLunches.com before proceeding to the next lab in the sequence.

In this chapter we're going to build on the functions you created in the last chapter using the concepts you hopefully picked up today. As you work through these labs, add verbose messages to display key steps or progress information.

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 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:

```
VERBOSE: Starting Get-Computerdata
VERBOSE: Getting data from localhost
VERBOSE: Win32_Computersystem
VERBOSE: Win32_Bios
VERBOSE: Win32_OperatingSystem

Workgroup               :
Manufacturer            : innotek GmbH
Computername            : CLIENT2
Version                 : 6.1.7601
Model                   : VirtualBox
AdminPassword           : NA
ServicePackMajorVersion : 1
BIOSSerial              : 0

VERBOSE: Ending Get-Computerdata
```

Here is a possible solution

```
Function Get-ComputerData {

[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName
 )

Begin {
    Write-Verbose “Starting Get-Computerdata”
 }

Process {
     foreach ($computer in $computerName) {
        Write-Verbose “Getting data from $computer”
        Write-Verbose “Win32_Computersystem”
            $cs = Get-WmiObject -Class Win32_Computersystem -ComputerName $Computer

            #decode the admin password status
            Switch ($cs.AdminPasswordStatus) {
                1 { $aps=”Disabled” }
                2 { $aps=”Enabled” }
                3 { $aps=”NA” }
                4 { $aps=”Unknown” }
            }
            #Define a hashtable to be used for property names and values
            $hash=@{
                Computername=$cs.Name
                Workgroup=$cs.WorkGroup
                AdminPassword=$aps
                Model=$cs.Model
                Manufacturer=$cs.Manufacturer
            }

            Write-Verbose “Win32_Bios”
            $bios = Get-WmiObject -Class Win32_Bios -ComputerName $Computer

            $hash.Add(“SerialNumber”,$bios.SerialNumber)

            Write-Verbose “Win32_OperatingSystem”
            $os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName
   $Computer
```

```
            $hash.Add("Version",$os.Version)
            $hash.Add("ServicePackMajorVersion",$os.ServicePackMajorVersion)

            #create a custom object from the hash table
            New-Object -TypeName PSObject -Property $hash

    } #foreach

} #process

End {
    Write-Verbose "Ending Get-Computerdata"
 }
}

"localhost" | Get-Computerdata -verbose
```

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:

```
VERBOSE: Starting Get-VolumeInfo
VERBOSE: Getting volume data from localhost
VERBOSE: Procssing volume \\?\Volume{8130d5f3-8e9b-11de-b460-806e6f6e6963}\

FreeSpace    Drive                    Computername       Size
---------    -----                    ------------       ----
0.07         \\?\Volume{8130d5f3...   CLIENT2            0.10
VERBOSE: Procssing volume C:\Temp\
9.78         C:\Temp\                 CLIENT2            10.00
VERBOSE: Procssing volume C:\
2.72         C:\                      CLIENT2            19.90
VERBOSE: Procssing volume D:\
2.72         D:\                      CLIENT2            4.00
VERBOSE: Ending Get-VolumeInfo
```

Here is a sample solution:

```
Function Get-VolumeInfo {

[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
```

```
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName
 )

Begin {
    Write-Verbose “Starting Get-VolumeInfo”
 }

Process {
   foreach ($computer in $computerName) {
    Write-Verbose “Getting volume data from $computer”
    $data = Get-WmiObject -Class Win32_Volume -computername $Computer -Filter
   “DriveType=3”

    Foreach ($drive in $data) {
      Write-Verbose “Procssing volume $($drive.name)”
        #format size and freespace
        $Size=”{0:N2}” -f ($drive.capacity/1GB)
        $Freespace=”{0:N2}” -f ($drive.Freespace/1GB)

        #Define a hashtable to be used for property names and values
        $hash=@{
            Computername=$drive.SystemName
            Drive=$drive.Name
            FreeSpace=$Freespace
            Size=$Size
        }

        #create a custom object from the hash table
        New-Object -TypeName PSObject -Property $hash
    } #foreach

    #clear $data for next computer
    Remove-Variable -Name data

} #foreach computer

} #Process

End {
    Write-Verbose “Ending Get-VolumeInfo”
 }
}

“localhost” | Get-VolumeInfo -verbose
```

Lab C

erName 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:

```
VERBOSE: Starting Get-ServiceInfo
VERBOSE: Getting services from localhost
VERBOSE: Processing service AudioEndpointBuilder

Computername       : CLIENT2
ThreadCount        : 13
ProcessName        : svchost.exe
Name               : AudioEndpointBuilder
VMSize             : 172224512
PeakPageFile       : 83112
Displayname        : Windows Audio Endpoint Builder
```

Here is a sample solution:

```
Function Get-ServiceInfo {

[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName
 )

 Begin {
    Write-Verbose "Starting Get-ServiceInfo"
 }

 Process {

   foreach ($computer in $computerName) {
   Write-Verbose "Getting services from $computer"
    $data = Get-WmiObject -Class Win32_Service -computername $Computer -Filter
   "State='Running'"

    foreach ($service in $data) {
      Write-Verbose "Processing service $($service.name)"
        $hash=@{
        Computername=$data[0].Systemname
        Name=$service.name
        Displayname=$service.DisplayName
```

```
        }

        #get the associated process
        $process=Get-WMIObject -class Win32_Process -computername $Computer
   -Filter "ProcessID='$($service.processid)'"
        $hash.Add("ProcessName",$process.name)
        $hash.add("VMSize",$process.VirtualSize)
        $hash.Add("PeakPageFile",$process.PeakPageFileUsage)
        $hash.add("ThreadCount",$process.Threadcount)

        #create a custom object from the hash table
        New-Object -TypeName PSObject -Property $hash

    } #foreach service

  } #foreach computer
} #process

End {
    Write-Verbose "Ending Get-ServiceInfo"
 }

}

"localhost" | Get-ServiceInfo -verbose
```

Standalone Lab

Use this script as your starting point:

```
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.Last-
   BootupTime));
                       '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 for should look something like this:

```
VERBOSE      : Getting WMI data from localhost

Model              : VirtualBox
ComputerName       : localhost
Manufacturer       : innotek GmbH
LastBootTime       : 6/19/2012 8:55:34 AM
OSVersion          : 6.1.760
```

Here is a sample solution:

```
function Get-SystemInfo {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName
    )
    PROCESS {
        foreach ($computer in $computerName) {
            Write-Verbose "Getting WMI data from $computer"
            $os = Get-WmiObject -class Win32_OperatingSystem -computerName
   $computer
            $cs = Get-WmiObject -class Win32_ComputerSystem -computerName $com-
   puter
            $props = @{'ComputerName'=$computer
                       'LastBootTime'=($os.ConvertToDateTime($os.Last-
   BootupTime))
                       'OSVersion'=$os.version
                       'Manufacturer'=$cs.manufacturer
                       'Model'=$cs.model
            }
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
}

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

Chapter 9

Lab Answers

Writing Help

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on MoreLunches.com before proceeding to the next lab in the sequence.

These labs will build on what you've already created, applying new concepts from this chapter.

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.

Here is a possible solution:

```
Function Get-ComputerData {

<#
.SYNOPSIS
Get computer related data

.DESCRIPTION

This command will query a remote computer and return a custom object with sys-
   tem information pulled from WMI. Depending on the computer some information
   may not be available.

.PARAMETER Computername

The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.

.EXAMPLE
PS C:\> Get-ComputerData Server01

Run the command and query Server01.

.EXAMPLE
```

```
PS C:\> get-content c:\work\computers.txt | Get-ComputerData

This expression will go through a list of computernames and pipe each name to
   the command.
#>

[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName
 )

  Begin {
    Write-Verbose “Starting Get-Computerdata”
 }

Process {
    foreach ($computer in $computerName) {
        Write-Verbose “Getting data from $computer”
        Write-Verbose “Win32_Computersystem”
            $cs = Get-WmiObject -Class Win32_Computersystem -ComputerName $Com-
   puter

            #decode the admin password status
            Switch ($cs.AdminPasswordStatus) {
                1 { $aps=”Disabled” }
                2 { $aps=”Enabled” }
                3 { $aps=”NA” }
                4 { $aps=”Unknown” }
            }

            #Define a hashtable to be used for property names and values
            $hash=@{
                Computername=$cs.Name
                Workgroup=$cs.WorkGroup
                AdminPassword=$aps
                Model=$cs.Model
                Manufacturer=$cs.Manufacturer
            }

            Write-Verbose “Win32_Bios”
            $bios = Get-WmiObject -Class Win32_Bios -ComputerName $Computer
```

```
            $hash.Add("SerialNumber",$bios.SerialNumber)

            Write-Verbose "Win32_OperatingSystem"
            $os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName
   $Computer
            $hash.Add("Version",$os.Version)
            $hash.Add("ServicePackMajorVersion",$os.ServicePackMajorVersion)

            #create a custom object from the hash table
            New-Object -TypeName PSObject -Property $hash

    } #foreach
} #process

 End {
    Write-Verbose "Ending Get-Computerdata"
 }
}

help Get-Computerdata -full
```

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 –ComputerName parameter. Test your help by adding help <function-name> to the end of your script.

Here is a possible solution:

```
Function Get-VolumeInfo {

<#
.SYNOPSIS
Get information about fixed volumes

.DESCRIPTION
This command will query a remote computer and return information about fixed vol-
   umes. The function will ignore network, optical and other removable drives.


.PARAMETER Computername
The name of a computer to query. The account you use to run this function
should have admin rights on that computer.

.EXAMPLE
PS C:\> Get-VolumeInfo Server01
```

```
Run the command and query Server01.

.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-VolumeInfo

This expression will go through a list of computernames and pipe each name to
   the command.

#>
[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName
 )

Begin {
    Write-Verbose “Starting Get-VolumeInfo”
 }

Process {
    foreach ($computer in $computerName) {

    $data = Get-WmiObject -Class Win32_Volume -computername $Computer -Filter
   “DriveType=3”

    Foreach ($drive in $data) {

        #format size and freespace
        $Size=”{0:N2}” -f ($drive.capacity/1GB)
        $Freespace=”{0:N2}” -f ($drive.Freespace/1GB)

        #Define a hashtable to be used for property names and values
        $hash=@{
            Computername=$drive.SystemName

            Drive=$drive.Name
            FreeSpace=$Freespace
            Size=$Size
        }

        #create a custom object from the hash table
        New-Object -TypeName PSObject -Property $hash
    } #foreach
```

```
    #clear $data for next computer
    Remove-Variable -Name data

  } #foreach computer
}#Process

 End {
    Write-Verbose “Ending Get-VolumeInfo”
 }
}

help Get-VolumeInfo -full
```

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.

Here is a possible solution:

```
Function Get-ServiceInfo {


<#
.SYNOPSIS
Get service information

.DESCRIPTION
This command will query a remote computer for running services and
   write a custom object to the pipeline that includes service details
   as well as a few key properties from the associated process. You
   must run this command with credentials that have admin rights on
   any remote computers.

.PARAMETER Computername
The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.

.EXAMPLE
PS C:\> Get-ServiceInfo Server01

Run the command and query Server01.

.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-ServiceInfo
```

```
This expression will go through a list of computernames and pipe each name to
   the command.

#>

[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName
 )

 Begin {
    Write-Verbose “Starting Get-ServiceInfo”
 }

 Process {

  foreach ($computer in $computerName) {
    $data = Get-WmiObject -Class Win32_Service -computername $Computer -Filter
   “State='Running'”

    foreach ($service in $data) {

        $hash=@{
        Computername=$data[0].Systemname
        Name=$service.name
        Displayname=$service.DisplayName
        }

        #get the associated process
        $process=Get-WMIObject -class Win32_Process -computername $Computer
   -Filter “ProcessID='$($service.processid)'”
        $hash.Add(“ProcessName”,$process.name)
        $hash.add(“VMSize”,$process.VirtualSize)
        $hash.Add(“PeakPageFile”,$process.PeakPageFileUsage)
        $hash.add(“ThreadCount”,$process.Threadcount)

        #create a custom object from the hash table
        New-Object -TypeName PSObject -Property $hash

    } #foreach service

   } #foreach computer
} #process

End {
    Write-Verbose “Ending Get-ServiceInfo”
 }
}

help Get-ServiceInfo -full
```

Standalone Lab

Using the script in Listing 9.2 add comment-based help.

List 9.2 Standalone lab starting point

```
function Get-SystemInfo {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName
    )
    PROCESS {
        foreach ($computer in $computerName) {
            Write-Verbose “Getting WMI data from $computer”
            $os = Get-WmiObject -class Win32_OperatingSystem -computerName
   $computer
            $cs = Get-WmiObject -class Win32_ComputerSystem -computerName $com-
   puter
            $props = @{‘ComputerName’=$computer
                       ‘LastBootTime’=($os.ConvertToDateTime($os.Last-
   BootupTime))
                       ‘OSVersion’=$os.version
                       ‘Manufacturer’=$cs.manufacturer
                       ‘Model’=$cs.model
                   }
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
}

Include at least a synopsis, description, and help for the -ComputerName param-
eter. Test your help by adding help <function-name> to the end of your script.
```

Here is a possible solution:

```
function Get-SystemInfo {
<#
.SYNOPSIS
Gets critical system info from one or more computers.
.DESCRIPTION
This command uses WMI, and can accept computer names, CNAME aliases, and IP ad-
   dresses. WMI must be enabled and you must run this with admin rights for any
   remote computer.
.PARAMETER Computername
One or more names or IP addresses to query.
.EXAMPLE
```

```
Get-SystemInfo -computername localhost
#>
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName
    )

    PROCESS {
        foreach ($computer in $computerName) {
            WWrite-Verbose “Getting WMI data from $computer”
            $os = Get-WmiObject -class Win32_OperatingSystem -computerName
   $computer
            $cs = Get-WmiObject -class Win32_ComputerSystem -computerName $com-
   puter
            $props = @{‘ComputerName’=$computer
                       ‘LastBootTime’=($os.ConvertToDateTime($os.Last-
   BootupTime))
                       ‘OSVersion’=$os.version
                       ‘Manufacturer’=$cs.manufacturer
                       ‘Model’=$cs.model
                     }
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
}

help Get-SystemInfo
```

Chapter
10

Lab Answers

Error Handling

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on MoreLunches.com before proceeding to the next lab in the sequence.

You are going to continue with the functions you've been building the last few chapters. The next step is to begin incorporating some error handling using Try/Catch/Finally. If you haven't done so, take a few minutes to read the help content on Try/Catch/Finally. For any changes you make, don't forget to update your comment-based help.

Lab A

Using Lab A from Chapter 9, add a –ErrorLog parameter to your advanced function, which accepts a filename for an error log and defaults to C:\Errors.txt. When the function is run with this parameter, failed computer names should be appended to the error log file.

Next, if the first WMI query fails, the function should output nothing for that computer and should not attempt a second or third WMI query. Write an error to the pipeline containing each failed computer name.

Test all of this by adding this line <function-name> -ComputerName localhost,NOTONLINE –verbose to the end of your script. A portion of the output should look something like this:

```
VERBOSE: Starting Get-Computerdata
VERBOSE: Getting data from localhost
VERBOSE: Win32_Computersystem
VERBOSE: Win32_Bios
VERBOSE: Win32_OperatingSystem

Workgroup                 :
Manufacturer              : innotek GmbH
Computername              : CLIENT2
Version                         : 6.1.7601
SerialNumber              : 0
Model                           : VirtualBox
AdminPassword             : NA
ServicePackMajorVersion   : 1
```

```
VERBOSE: Getting data from notonline
VERBOSE: Win32_Computersystem
Get-Computerdata : Failed getting system information from notonline. The RPC
   server is
unavailable. (Exception from HRESULT: 0x800706BA)
At S:\Toolmaking\Ch10-LabA.ps1:115 char:40
+ 'localhost','notonline','localhost' |  Get-Computerdata -logerrors -verbose
+                                        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorExcep-
   tion
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.
   WriteErrorException,Get-Comp
   uterData


VERBOSE: Getting data from localhost
```

Here is a sample solution:

```
Function Get-ComputerData {


<#
.SYNOPSIS
Get computer related data


.DESCRIPTION
This command will query a remote computer and return a custom object with sys-
   tem information pulled from WMI. Depending on the computer some information
   may not be available.


.PARAMETER Computername
The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.


.PARAMETER ErrorLog
Specify a path to a file to log errors. The default is C:\Errors.txt


.EXAMPLE
PS C:\> Get-ComputerData Server01


Run the command and query Server01.


.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-ComputerData -Errorlog c:\logs\
   errors.txt


This expression will go through a list of computernames and pipe each name to
   the command. Computernames that can't be accessed will be written to the log
   file.
```

```
#>

[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName,
 [string]$ErrorLog="C:\Errors.txt"
 )

 Begin {
    Write-Verbose "Starting Get-Computerdata"
 }

Process {
    foreach ($computer in $computerName) {
        Write-Verbose "Getting data from $computer"
        Try {
            Write-Verbose "Win32_Computersystem"
            $cs = Get-WmiObject -Class Win32_Computersystem -ComputerName $Com-
   puter -ErrorAction Stop

            #decode the admin password status
            Switch ($cs.AdminPasswordStatus) {
                1 { $aps="Disabled" }
                2 { $aps="Enabled" }
                3 { $aps="NA" }
                4 { $aps="Unknown" }
            }

            #Define a hashtable to be used for property names and values
            $hash=@{

                Computername=$cs.Name
                Workgroup=$cs.WorkGroup
                AdminPassword=$aps
                Model=$cs.Model
                Manufacturer=$cs.Manufacturer
            }

        } #Try

        Catch {
```

```
            #create an error message
            $msg="Failed getting system information from $computer. $($_.Excep-
    tion.Message)"
            Write-Error $msg

            Write-Verbose "Logging errors to $errorlog"
            $computer | Out-File -FilePath $Errorlog -append

             } #Catch

        #if there were no errors then $hash will exist and we can continue and
    assume
        #all other WMI queries will work without error
        If ($hash) {
            Write-Verbose "Win32_Bios"
            $bios = Get-WmiObject -Class Win32_Bios -ComputerName $Computer
            $hash.Add("SerialNumber",$bios.SerialNumber)

            Write-Verbose "Win32_OperatingSystem"
            $os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName
    $Computer
            $hash.Add("Version",$os.Version)
            $hash.Add("ServicePackMajorVersion",$os.ServicePackMajorVersion)

            #create a custom object from the hash table
            New-Object -TypeName PSObject -Property $hash

            #remove $hash so it isn't accidentally re-used by a computer that
    causes
            #an error

            Remove-Variable -name hash
        } #if $hash
     } #foreach
} #process

 End {
    Write-Verbose "Ending Get-Computerdata"
 }
}

'localhost','notonline','localhost' |  Get-Computerdata  -verbose
```

Lab B

a filename for an error log and defaults to C:\Errors.txt. When the function is run with this parameter, failed computer names should be appended to the error log file.

Test all of this by adding this line <function-name> -ComputerName localhost,NOTONLINE –verbose to the end of your script. A portion of the output should look something like this:

```
VERBOSE: Starting Get-VolumeInfo
VERBOSE: Getting data from localhost

FreeSpace          Drive                      Computername       Size
---------          -----                      ------------       ----
0.07               \\?\Volume{8130d5f3...     CLIENT2            0.10
9.78               C:\Temp\                   CLIENT2            10.00
2.72               C:\                        CLIENT2            19.90
2.72               D:\                        CLIENT2            4.00
VERBOSE: Getting data from NotOnline
Get-VolumeInfo : Failed to get volume information from NotOnline. The RPC serv-
   er is
unavailable. (Exception from HRESULT: 0x800706BA)
At S:\Toolmaking\Ch10-LabB.ps1:96 char:27
+ 'localhost','NotOnline' | Get-VolumeInfo -Verbose -logerrors
+                           ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorExcep-
   tion
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.
   WriteErrorException,Get-VolumeInfo

VERBOSE: Logging errors to C:\Errors.txt
VERBOSE: Ending Get-VolumeInfo
```

Here is a sample solution:

```
Function Get-VolumeInfo {

<#
.SYNOPSIS
Get information about fixed volumes

.DESCRIPTION
This command will query a remote computer and return information about fixed vol-
   umes. The function will ignore network, optical and other removable drives.
.PARAMETER Computername
The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.

.PARAMETER ErrorLog
```

```
Specify a path to a file to log errors. The default is C:\Errors.txt

.EXAMPLE
PS C:\> Get-VolumeInfo Server01

Run the command and query Server01.

.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-VolumeInfo -errorlog c:\logs\
   errors.txt

This expression will go through a list of computernames and pipe each name to
   the command. Computernames that can't be accessed will be written to the log
   file.

#>

[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName,
 [string]$ErrorLog="C:\Errors.txt",
  [switch]$LogErrors
 )

Begin {
    Write-Verbose "Starting Get-VolumeInfo"
 }

Process {
    foreach ($computer in $computerName) {
        Write-Verbose "Getting data from $computer"
        Try {
            $data = Get-WmiObject -Class Win32_Volume -computername $Computer
   -Filter "DriveType=3" -ErrorAction Stop

            Foreach ($drive in $data) {
                   Write-Verbose "Processing volume $($drive.name)"
                #format size and freespace
                $Size="{0:N2}" -f ($drive.capacity/1GB)
                $Freespace="{0:N2}" -f ($drive.Freespace/1GB)

                #Define a hashtable to be used for property names and values
                $hash=@{
```

```
                    Computername=$drive.SystemName
                    Drive=$drive.Name
                    FreeSpace=$Freespace
                    Size=$Size
                }

                #create a custom object from the hash table

                New-Object -TypeName PSObject -Property $hash
            } #foreach

            #clear $data for next computer
            Remove-Variable -Name data

        } #Try

        Catch {
            #create an error message
            $msg=”Failed to get volume information from $computer. $($_.Excep-
   tion.Message)”
            Write-Error $msg

            Write-Verbose “Logging errors to $errorlog”
            $computer | Out-File -FilePath $Errorlog -append
        }
    } #foreach computer
} #Process

 End {
    Write-Verbose “Ending Get-VolumeInfo”
 }
}

‘localhost’,’NotOnline’ | Get-VolumeInfo -Verbose
```

Lab C

Using Lab C from Chapter 9, add a –LogErrors switch parameter to your advanced function. Also add a –ErrorFile parameter, which accepts a filename for an error log and defaults to C:\Errors.txt. When the function is run with the -LogErrors parameter, failed computer names should be appended to the error log file. .Also, if –LogErrors is used, the log file should be deleted at the start of the function if it exists, so that each time the command starts with a fresh log file.

Test all of this by adding this line <function-name> -ComputerName localhost,NOTONLINE –verbose –logerrors to the end of your script. A portion of the output should look something like this:

```
VERBOSE: Processing service wuauserv
VERBOSE: Getting process for wuauserv
Computername : CLIENT2
ThreadCount  : 45
ProcessName  : svchost.exe
Name         : wuauserv
VMSize       : 499363840
PeakPageFile : 247680
Displayname  : Windows Update

VERBOSE: Getting services from NOTOnline
Get-ServiceInfo : Failed to get service data from NOTOnline. The RPC server is
unavailable. (Exception from HRESULT: 0x800706BA)
At S:\Toolmaking\Ch10-LabC.ps1:109 char:39
+ “localhost”,”NOTOnline”,”localhost” | Get-ServiceInfo -logerrors -verbose
+                                       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorExcep-
   tion
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.
   WriteErrorException,Get-ServiceInfo

VERBOSE: Logging errors to C:\Errors.txt
VERBOSE: Getting services from localhost
VERBOSE: Processing service AudioEndpointBuilder
VERBOSE: Getting process for AudioEndpointBuilder
```

Here is a sample solution:

```
Function Get-ServiceInfo {

<#
.SYNOPSIS
Get service information
```

```
.DESCRIPTION
This command will query a remote computer for running services and write a cus-
   tom object to the pipeline that includes service details as well as a few
   key properties from the associated process. You must run this command with
   credentials that have admin rights on any remote computers.


.PARAMETER Computername
The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.


.PARAMETER ErrorLog
Specify a path to a file to log errors. The default is C:\Errors.txt


.PARAMETER LogErrors
If specified, computer names that can't be accessed will be logged to the file
   specified by -Errorlog.


.EXAMPLE
PS C:\> Get-ServiceInfo Server01


Run the command and query Server01.


.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-ServiceInfo -logerrors


This expression will go through a list of computernames and pipe each name to
   the command. Computernames that can't be accessed will be written to the log
   file.


#>


[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName,
 [string]$ErrorLog="C:\Errors.txt",
 [switch]$LogErrors
 )

 Begin {
    Write-Verbose "Starting Get-ServiceInfo"

    #if -LogErrors and error log exists, delete it.
    if ( (Test-Path -path $errorLog) -AND $LogErrors) {
        Write-Verbose "Removing $errorlog"
```

```
        Remove-Item $errorlog
    }
}

Process {

    foreach ($computer in $computerName) {
        Write-Verbose "Getting services from $computer"

        Try {
            $data = Get-WmiObject -Class Win32_Service -computername $Computer
  -Filter "State='Running'" -ErrorAction Stop

            foreach ($service in $data) {
                Write-Verbose "Processing service $($service.name)"
                $hash=@{
                Computername=$data[0].Systemname
                Name=$service.name
                Displayname=$service.DisplayName
                }

                #get the associated process
                Write-Verbose "Getting process for $($service.name)"
                $process=Get-WMIObject -class Win32_Process -computername $Com-
  puter -Filter "ProcessID='$($service.processid)'" -ErrorAction Stop
                $hash.Add("ProcessName",$process.name)
                $hash.add("VMSize",$process.VirtualSize)
                $hash.Add("PeakPageFile",$process.PeakPageFileUsage)
                $hash.add("ThreadCount",$process.Threadcount)

                #create a custom object from the hash table
                New-Object -TypeName PSObject -Property $hash

            } #foreach service

            }
        Catch {
            #create an error message
            $msg="Failed to get service data from $computer. $($_.Exception.
  Message)"
            Write-Error $msg

            if ($LogErrors) {
                Write-Verbose "Logging errors to $errorlog"
                 $computer | Out-File -FilePath $Errorlog -append
            }
```

```
        }

    } #foreach computer

} #process

End {
    Write-Verbose “Ending Get-ServiceInfo”
 }

}

Get-ServiceInfo -ComputerName “localhost”,”NOTOnline”,”localhost”  -logerrors
```

Standalone Lab

Use the code in Listing 10.4 as a starting point.

Listing 10.4 Standalone lab starting point

```
Function Get-SystemInfo {

<#
.SYNOPSIS
Gets critical system info from one or more computers.
.DESCRIPTION
This command uses WMI, and can accept computer names, CNAME aliases, and IP ad-
   dresses. WMI must be enabled and you must run this
with admin rights for any remote computer.
.PARAMETER Computername
One or more names or IP addresses to query.
.EXAMPLE
Get-SystemInfo -computername localhost
#>
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName
    )
    PROCESS {
        foreach ($computer in $computerName) {
            WWrite-Verbose “Getting WMI data from $computer”
            $os = Get-WmiObject -class Win32_OperatingSystem -computerName
   $computer
```

```
            $cs = Get-WmiObject -class Win32_ComputerSystem -computerName $com-
   puter
            $props = @{'ComputerName'=$computer
                       'LastBootTime'=($os.ConvertToDateTime($os.Last-
   BootupTime))
                       'OSVersion'=$os.version
                       'Manufacturer'=$cs.manufacturer
                       'Model'=$cs.model
                      }
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    }
}
```

Add a –LogErrors switch to this advanced function. When the function is run with this switch, failed computer names should be logged to C:\Errors.txt. This file should be deleted at the start of the function each time it is run, so that it starts out fresh each time. If the first WMI query fails, the function should output nothing for that computer and should not attempt a second WMI query. Write an error to the pipeline containing each failed computer name.

Test your script by adding this line to the end of your script.

```
Get-SystemInfo -computername localhost,NOTONLINE,localhost -logerrors
A portion of the output should look something like this:
Model        : VirtualBox
ComputerName : localhost
Manufacturer : innotek GmbH
LastBootTime : 6/19/2012 8:55:34 AM
OSVersion    : 6.1.7601

Get-SystemInfo : NOTONLINE failed
At S:\Toolmaking\Ch10-Standalone.ps1:51 char:1
+ Get-SystemInfo -computername localhost,NOTONLINE,localhost -logerrors
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorExcep-
   tion
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.
   WriteErrorException,Get-Syst
   emInfo

Model        : VirtualBox
ComputerName : localhost
Manufacturer : innotek GmbH
LastBootTime : 6/19/2012 8:55:34 AM
OSVersion    : 6.1.7601
```

Here is a sample solution:

```
function Get-SystemInfo {
<#
.SYNOPSIS
Gets critical system info from one or more computers.
.DESCRIPTION
This command uses WMI, and can accept computer names, CNAME aliases, and IP ad-
   dresses. WMI must be enabled and you must run this with admin rights for any
   remote computer.
.PARAMETER Computername
One or more names or IP addresses to query.
.EXAMPLE
Get-SystemInfo -computername localhost
#>
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName,
        [switch]$logErrors
    )
    BEGIN {
        if (Test-Path c:\errors.txt) {
            del c:\errors.txt
        }
    }
    PROCESS {
        foreach ($computer in $computerName) {
            WWrite-Verbose “Getting WMI data from $computer”
            try {
                $continue = $true
                $os = Get-WmiObject -class Win32_OperatingSystem -computerName
   $computer -ErrorAction Stop
            } catch {
                $continue = $false
                $computer | Out-File c:\errors.txt -append
                Write-Error “$computer failed”

            }
            if ($continue) {
                $cs = Get-WmiObject -class Win32_ComputerSystem   -computerName
   $computer
                $props = @{‘ComputerName’=$computer
                           ‘LastBootTime’=($os.ConvertToDateTime($os.Last-
   BootupTime))
```

```
                            'OSVersion'=$os.version
                            'Manufacturer'=$cs.manufacturer
                            'Model'=$cs.model
                        }
                $obj = New-Object -TypeName PSObject -Property $props
                Write-Output $obj
            }
        }
    }
}

Get-SystemInfo -computername localhost,NOTONLINE,localhost -logerrors
```

Chapter
11

Lab Answers

Debugging Techniques

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

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

The function in Listing 11.8 is supposed to display some properties of running services sorted by the service account.

The function in listing 11.9 is a bit more involved. It's designed to get recent event log entries for a specified log on a specified computer. Events are sorted by the event source and added to a log file. The filename is based on the date, computer name, and event source. At the end, the function displays a directory listing of the logs. Hint: Clean up the formatting first.

Lab A

Listing 11.8 A broken function

```
1  Function Get-ServiceInfo {
2  [cmdletbinding()]
3  Param([string]$Computername)
4  $services=Get-WmiObject -Class Win32_Services -filter “state=’Running” `
    -computername $computernam
5  Write-Host “Found ($services.count) on $computername” –Foreground Green
6  $sevices | sort -Property startname,name  Select -property `
   startname,name,startmode,computername
7  }
```

Lab B

Listing 11.9 Buggy Export Function

```
01  Function Export-EventLogSource {
02
03  [cmdletbinding()]
04  Param (
05  [Parameter(Position=0,Mandatory=$True,Helpmessage="Enter a computername",Va
    lueFromPipeline=$True)]
06  [string]$Computername,
07  [Parameter(Position=1,Mandatory=$True,Helpmessage="Enter a classic event
    log name like System")]
08  [string]$Log,
09  [int]$Newest=100
10  )
11  Begin {
12  Write-Verbose "Starting export event source function"
13  #the date format is case-sensitive"
14  $datestring=Get-Date -Format "yyyyMMdd"
15  $logpath=Join-path -Path "C:\Work" -ChildPath $datestring
16  if (! (Test-Path -path $logpath) {
17  Write-Verbose "Creating $logpath"
18  mkdir $logpath
19  }
20  Write-Verbose "Logging results to $logpath"
21  }
22  Process {
23  Write-Verbose "Getting newest $newest $log event log entries from $comput-
    ername"
24  Try {
25  Write-Host $computername.ToUpper -ForegroundColor Green
26  $logs=Get-EventLog -LogName $log -Newest $Newest -Computer $Computer -Er-
    rorAction Stop
27  if ($logs) {
28  Write-Verbose "Sorting $($logs.count) entries"
29  $log | sort Source | foreach {
30  $logfile=Join-Path -Path $logpath -ChildPath "$computername-$($_.Source).
    txt"
31  $_ | Format-List TimeWritten,MachineName,EventID,EntryType,Message |
32  Out-File -FilePath $logfile -append
33
34  #clear variables for next time
35  Remove-Variable -Name logs,logfile
36  }
37  else {Write-Warning "No logged events found for $log on $Computername"}
38  }
39   Catch { Write-Warning $_.Exception.Message }
```

```
40  }
41  End {dir $logpath
42  Write-Verbose “Finished export event source function”
43  }
44  }
```

Answer

```
01  Function Export-EventLogSource {
02
03  [cmdletbinding()]
04
05  Param (
06  [Parameter(Position=0,Mandatory=$True,Helpmessage=”Enter a computername”,Va
    lueFromPipeline=$True)]
07  [string]$Computername,
08  [Parameter(Position=1,Mandatory=$True,Helpmessage=”Enter a classic event
    log name like System”)]
09  [string]$Log,
10  [int]$Newest=100
11  )
12
13  Begin {
14      Write-Verbose “Starting export event source function”
15
16      #the date format is case-sensitive”
17      $datestring=Get-Date -Format “yyyyMMdd”
18      $logpath=Join-path -Path “C:\Work” -ChildPath $datestring
19
20      if (! (Test-Path -path $logpath) {
21        Write-Verbose “Creating $logpath”
22        mkdir $logpath
23  }
24
25  Write-Verbose “Logging results to $logpath”
26
27  }
28
29  Process {
30      Write-Verbose “Getting newest $newest $log event log entries from $com-
    putername”
31
32      Try {
```

```
33          Write-Host $computername.ToUpper -ForegroundColor Green
34          $logs=Get-EventLog -LogName $log -Newest $Newest -Computer $Comput-
er -ErrorAction Stop
35          if ($logs) {
36              Write-Verbose “Sorting $($logs.count) entries”
37              $log | sort Source | foreach {
38              $logfile=Join-Path -Path $logpath -ChildPath “$computername-$($_.
Source).txt”
39              $_ | Format-List TimeWritten,MachineName,EventID,EntryType,Mess
age | Out-File -FilePath $logfile -append
40
41              #clear variables for next time
42              Remove-Variable -Name logs,logfile
43          }
44          else {
45              Write-Warning “No logged events found for $log on $Computer-
name”
46          }
47      }
48       Catch {
49          Write-Warning $_.Exception.Message
50      }
51  }
52
53  End {
54      dir $logpath
55      Write-Verbose “Finished export event source function”
56  }
57  }
```

Chapter
12

Lab Answers

Creating Custom Format Views

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on More-Lunches.com before proceeding to the next lab in the sequence.

We bet you can guess what is coming. You'll be adding type information and creating custom format files for the functions you've been working on the last several chapters. Use the dotnettypes.format.ps1xml and other .ps1xml files as sources for sample layout. Copy and paste the XML into your new format file. Don't forget that tags are case-sensitive.

Lab A

Modify your advanced function from Lab A in Chapter 10 so that the output object has the type name MOL.ComputerSystemInfo. Then, create a custom view in a file named C:\CustomViewA.format.ps1xml. The custom view should display objects of the type MOL.ComputerSystemInfo in a list format, displaying the information in a list as indicated in your design for this lab. Go back to Chapter 6 to check what the output names should be.

```
At the bottom of the script file, add these commands to test:
Update-FormatData -prepend c:\CustomViewA.format.ps1xml
<function-name> -ComputerName localhost
```

The final output should look something like the following.

```
Computername          : CLIENT2
Workgroup             :
AdminPassword         : NA
Model                 : VirtualBox
Manufacturer          : innotek GmbH
BIOSSerialNumber      : 0
OSVersion             : 6.1.7601
SPVersion             : 1
```

Note that the list labels are not exactly the same as the custom object's property names.

Sample format file

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
    <ViewDefinitions>
        <View>
            <Name>MOL.SystemInfo</Name>
            <ViewSelectedBy>
                <TypeName>MOL.ComputerSystemInfo</TypeName>
            </ViewSelectedBy>
            <ListControl>
                <ListEntries>
                <ListEntry>
                    <ListItems>
                        <ListItem>
                            <PropertyName>ComputerName</PropertyName>
                        </ListItem>
                        <ListItem>
                            <PropertyName>Workgroup</PropertyName>
                        </ListItem>
                        <ListItem>
                            <PropertyName>AdminPassword</PropertyName>
                        </ListItem>
                        <ListItem>
                            <Propertyname>Model</Propertyname>
                        </ListItem>
                        <ListItem>
                            <Propertyname>Manufacturer</Propertyname>
                        </ListItem>
                        <ListItem>
                            <Propertyname>SerialNumber</Propertyname>
                            <Label>BIOSSerialNumber</Label>
                        </ListItem>
                        <ListItem>
                            <Propertyname>Version</Propertyname>
                            <Label>OSVersion</Label>
                        </ListItem>
                        <ListItem>
                            <Propertyname>ServicePackMajorVersion</Propertyname>
                            <Label>SPVersion</Label>
                        </ListItem>
                    </ListItems>
                </ListEntry>
               </ListEntries>
            </ListControl>
```

```
        </View>
    </ViewDefinitions>
</Configuration>
```

Sample Script

```
Function Get-ComputerData {

<#
.SYNOPSIS
Get computer related data

.DESCRIPTION
This command will query a remote computer and return a custom object with sys-
   tem information pulled from WMI. Depending on the computer some information
   may not be available.

.PARAMETER Computername
The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.

.PARAMETER ErrorLog
Specify a path to a file to log errors. The default is C:\Errors.txt

.EXAMPLE
PS C:\> Get-ComputerData Server01

Run the command and query Server01.

.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-ComputerData -Errorlog c:\logs\
   errors.txt
This expression will go through a list of computernames and pipe each name to
   the command. Computernames that can't be accessed will be written to the log
   file.

#>

[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName,
 [string]$ErrorLog="C:\Errors.txt"
 )
```

```
 Begin {
    Write-Verbose “Starting Get-Computerdata”
 }

Process {
    foreach ($computer in $computerName) {
        Write-Verbose “Getting data from $computer”
        Try {
            Write-Verbose “Win32_Computersystem”
            $cs = Get-WmiObject -Class Win32_Computersystem -ComputerName $Com-
   puter -ErrorAction Stop

            #decode the admin password status
            Switch ($cs.AdminPasswordStatus) {
                1 { $aps=”Disabled” }
                2 { $aps=”Enabled” }
                3 { $aps=”NA” }
                4 { $aps=”Unknown” }
            }

            #Define a hashtable to be used for property names and values
            $hash=@{
                Computername=$cs.Name
                Workgroup=$cs.WorkGroup
                AdminPassword=$aps
                Model=$cs.Model
                Manufacturer=$cs.Manufacturer
            }

        } #Try

        Catch {

            #create an error message
            $msg=”Failed getting system information from $computer. $($_.Excep-
   tion.Message)”
            Write-Error $msg

            Write-Verbose “Logging errors to $errorlog”
            $computer | Out-File -FilePath $Errorlog -append

            } #Catch

        #if there were no errors then $hash will exist and we can continue and
   assume
        #all other WMI queries will work without error
```

```
        If ($hash) {
            Write-Verbose “Win32_Bios”
            $bios = Get-WmiObject -Class Win32_Bios -ComputerName $Computer
            $hash.Add(“SerialNumber”,$bios.SerialNumber)

            Write-Verbose “Win32_OperatingSystem”
            $os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName
   $Computer
            $hash.Add(“Version”,$os.Version)
            $hash.Add(“ServicePackMajorVersion”,$os.ServicePackMajorVersion)

            #create a custom object from the hash table
            $obj=New-Object -TypeName PSObject -Property $hash
            #add a type name to the custom object
            $obj.PSObject.TypeNames.Insert(0,’MOL.ComputerSystemInfo’)

            Write-Output $obj
            #remove $hash so it isn’t accidentally re-used by a computer that
   causes
            #an error
            Remove-Variable -name hash
        } #if $hash
    } #foreach
} #process

 End {
    Write-Verbose “Ending Get-Computerdata”
 }
}

Update-FormatData –prepend C:\CustomViewA.format.ps1xml
Get-ComputerData -ComputerName localhost
```

Lab B

Modify your advanced function Lab B from Chapter 10 so that the output object has the type name MOL.DiskInfo. Then, create a custom view in a file named C:\CustomViewB.format.ps1xml. The custom view should display objects of the type MOL.DiskInfo in a table format, displaying the information in a table as indicated in your design for this lab. Refer back to Chapter 6 for a refresher. The column headers for the FreeSpace and Size properties should display "FreeSpace(GB)" and "Size(GB)," respectively.

At the bottom of the script file, add these commands to test:

```
Update-FormatData –prepend c:\CustomViewB.format.ps1xml
<function-name> -ComputerName localhost
```

The final output should look something like the following.

```
ComputerName       Drive                          FreeSpace(GB)   Size(GB)
------------       -----                          -------------   --------
CLIENT2            \\?\Volume{8130d5f3-8e9b-...   0.07            0.10
CLIENT2            C:\Temp\                       9.78            10.00
CLIENT2            C:\                            2.72            19.90
CLIENT2            D:\                            2.72            4.00
```

Note that the column headers are not exactly the same as the custom object's property names.

Sample format file solution

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
    <ViewDefinitions>
        <View>
            <Name>MOL.SystemInfo</Name>
            <ViewSelectedBy>
                <TypeName>MOL.DiskInfo</TypeName>
            </ViewSelectedBy>
            <TableControl>
                <TableHeaders>
                    <TableColumnHeader>
                    <Width>18</Width>
                    </TableColumnHeader>
                    <TableColumnHeader/>
                    <TableColumnHeader>
                        <Label>FreeSpace(GB)</Label>
                        <Width>15</Width>
                    </TableColumnHeader>
                    <TableColumnHeader>
                        <Label>Size(GB)</Label>
```

```
                        <Width>10</Width>
                    </TableColumnHeader>
                </TableHeaders>
                <TableRowEntries>
                    <TableRowEntry>
                        <TableColumnItems>
                            <TableColumnItem>
                                <PropertyName>ComputerName</PropertyName>
                            </TableColumnItem>
                            <TableColumnItem>
                                <PropertyName>Drive</PropertyName>
                            </TableColumnItem>
                            <TableColumnItem>
                                <PropertyName>FreeSpace</PropertyName>
                            </TableColumnItem>
                            <TableColumnItem>
                                <Propertyname>Size</Propertyname>
                            </TableColumnItem>
                        </TableColumnItems>
                    </TableRowEntry>
                </TableRowEntries>
            </TableControl>
        </View>
    </ViewDefinitions>
</Configuration>

Sample script solution
Function Get-VolumeInfo {

<#
.SYNOPSIS
Get information about fixed volumes

.DESCRIPTION
This command will query a remote computer and return information about fixed vol-
   umes. The function will ignore network, optical and other removable drives.

.PARAMETER Computername
The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.

.PARAMETER ErrorLog
Specify a path to a file to log errors. The default is C:\Errors.txt

.EXAMPLE
```

```
PS C:\> Get-VolumeInfo Server01

Run the command and query Server01.

.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-VolumeInfo -errorlog c:\logs\
   errors.txt

This expression will go through a list of computernames and pipe each name to
   the command. Computernames that can't be accessed will be written to the log
   file.

#>
[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName,
 [string]$ErrorLog="C:\Errors.txt",
  [switch]$LogErrors
 )

Begin {
    Write-Verbose "Starting Get-VolumeInfo"
 }

Process {
    foreach ($computer in $computerName) {
        Write-Verbose "Getting data from $computer"
        Try {
            $data = Get-WmiObject -Class Win32_Volume -computername $Computer
   -Filter "DriveType=3" -ErrorAction Stop

            Foreach ($drive in $data) {
                Write-Verbose "Processing volume $($drive.name)"
                #format size and freespace
                $Size="{0:N2}" -f ($drive.capacity/1GB)
                $Freespace="{0:N2}" -f ($drive.Freespace/1GB)

                #Define a hashtable to be used for property names and values
                $hash=@{
                    Computername=$drive.SystemName
                    Drive=$drive.Name
                    FreeSpace=$Freespace
                    Size=$Size
```

```
                }

                #create a custom object from the hash table
                $obj=New-Object -TypeName PSObject -Property $hash
                #Add a type name to the object
                $obj.PSObject.TypeNames.Insert(0,'MOL.DiskInfo')

                 Write-Output $obj

            } #foreach

            #clear $data for next computer
            Remove-Variable -Name data


        } #Try

        Catch {
            #create an error message
            $msg="Failed to get volume information from $computer. $($_.Excep-
   tion.Message)"
            Write-Error $msg

            Write-Verbose "Logging errors to $errorlog"
            $computer | Out-File -FilePath $Errorlog -append
        }
    } #foreach computer
} #Process

 End {
    Write-Verbose "Ending Get-VolumeInfo"
 }
}

Update-FormatData -prepend C:\CustomViewB.format.ps1xml
Get-VolumeInfo localhost
```

Lab C

Modify your advanced function Lab C from Chapter 10 so that the output object has the type name MOL.ServiceProcessInfo. Then, create a custom view in a file named C:\CustomViewC.format.ps1xml. The custom view should display objects of the type MOL.ServiceProcessInfo in a table format, displaying computername, service name, display name, process name, and process virtual size.

In addition to the table format, create a list view in the same file that displays the properties in this order:

- Computername
- Name (renamed as Service)
- Displayname
- ProcessName
- VMSize
- ThreadCount
- PeakPageFile

At the bottom of the script file, add these commands to test:

```
Update-FormatData –prepend c:\CustomViewC.format.ps1xml
<function-name> -ComputerName localhost
<function-name> -ComputerName localhost | Format-List
```

The final output should look something like this for the table.

```
ComputerName    Service          Displayname          ProcessName    VM
------------    -------          -----------          -----------    --
CLIENT2         AudioEndpo...    Windows Audio E...   svchost.exe    172208128
CLIENT2         BFE              Base Filtering ...   svchost.exe    69496832
CLIENT2         BITS             ackground Inte...    svchost.exe    499310592
CLIENT2         Browser          Computer Browser     svchost.exe    499310592
```

And like this for the list:

```
Computername : CLIENT2
Service      : AudioEndpointBuilder
Displayname  : Windows Audio Endpoint Builder
ProcessName  : svchost.exe
VMSize       : 172208128
ThreadCount  : 13
PeakPageFile : 83112
```

Note that per the design specifications from Chapter 6 not every object property is displayed by default and that some column headings are different than the actual property names.

Sample format file solution:

```
<?xml version=”1.0” encoding=”utf-8” ?>
<Configuration>
    <ViewDefinitions>
        <View>
            <Name>MOL.SystemInfo</Name>
```

```
<ViewSelectedBy>
    <TypeName>MOL.ServiceProcessInfo</TypeName>
</ViewSelectedBy>
<TableControl>
    <TableHeaders>
        <TableColumnHeader>
             <Width>14</Width>
        </TableColumnHeader>
        <TableColumnHeader>
             <Label>Service</Label>
             <Width>13</Width>
        </TableColumnHeader>
        <TableColumnHeader>
             <Width>18</Width>
        </TableColumnHeader>
        <TableColumnHeader>
             <Width>17</Width>
             </TableColumnHeader>
        <TableColumnHeader>
             <Label>VM</Label>
             <Width>14</Width>
        </TableColumnHeader>
    </TableHeaders>
    <TableRowEntries>
        <TableRowEntry>
            <TableColumnItems>
                <TableColumnItem>
                    <PropertyName>ComputerName</PropertyName>
                </TableColumnItem>
                <TableColumnItem>
                    <PropertyName>Name</PropertyName>
                </TableColumnItem>
                <TableColumnItem>
                    <PropertyName>Displayname</PropertyName>
                </TableColumnItem>
                <TableColumnItem>
                    <Propertyname>ProcessName</Propertyname>
                </TableColumnItem>
                <TableColumnItem>
                    <Propertyname>VMSize</Propertyname>
                </TableColumnItem>
            </TableColumnItems>
        </TableRowEntry>
     </TableRowEntries>
</TableControl>
```

```
            </View>
            <View>
             <Name>MOL.SystemInfo</Name>
             <ViewSelectedBy>
             <TypeName>MOL.ServiceProcessInfo</TypeName>
             </ViewSelectedBy>
                      <ListControl>
                      <ListEntries>
                 <ListEntry>
                     <ListItems>
                         <ListItem>
                             <PropertyName>ComputerName</PropertyName>
                         </ListItem>
                         <ListItem>
                             <PropertyName>Name</PropertyName>
                             <Label>Service</Label>
                         </ListItem>
                         <ListItem>
                             <PropertyName>Displayname</PropertyName>
                         </ListItem>
                         <ListItem>
                             <Propertyname>ProcessName</Propertyname>
                         </ListItem>
                         <ListItem>
                             <Propertyname>VMSize</Propertyname>
                         </ListItem>
                         <ListItem>
                             <Propertyname>ThreadCount</Propertyname>
                         </ListItem>
                         <ListItem>
                             <Propertyname>PeakPageFile</Propertyname>
                         </ListItem>
                     </ListItems>
                 </ListEntry>
                </ListEntries>
             </ListControl>
        </View>
    </ViewDefinitions>
</Configuration>
```

Sample script solution:

```
Function Get-ServiceInfo {

<#
.SYNOPSIS
Get service information

.DESCRIPTION
This command will query a remote computer for running services and write a cus-
   tom object to the pipeline that includes service details as well as a few
   key properties from the associated process. You must run this command with
   credentials that have admin rights on any remote computers.

.PARAMETER Computername
The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.

.PARAMETER ErrorLog
Specify a path to a file to log errors. The default is C:\Errors.txt

.PARAMETER LogErrors
If specified, computer names that can't be accessed will be logged  to the file
   specified by -Errorlog.

.EXAMPLE
PS C:\> Get-ServiceInfo Server01

Run the command and query Server01.



.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-ServiceInfo -logerrors

This expression will go through a list of computernames and pipe each name to
   the command. Computernames that can't be accessed will be written to the log
   file.

#>

[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName,
 [string]$ErrorLog="C:\Errors.txt",
 [switch]$LogErrors
```

```
)

Begin {
   Write-Verbose “Starting Get-ServiceInfo”

   #if -LogErrors and error log exists, delete it.
   if ( (Test-Path -path $errorLog) -AND $LogErrors) {
       Write-Verbose “Removing $errorlog”
       Remove-Item $errorlog
   }
}

Process {

   foreach ($computer in $computerName) {
       Write-Verbose “Getting services from $computer”

       Try {
           $data = Get-WmiObject -Class Win32_Service -computername $Computer
  -Filter “State=’Running’” -ErrorAction Stop

           foreach ($service in $data) {
               Write-Verbose “Processing service $($service.name)”
               $hash=@{
               Computername=$data[0].Systemname
               Name=$service.name
               Displayname=$service.DisplayName
               }

               #get the associated process
               Write-Verbose “Getting process for $($service.name)”
               $process=Get-WMIObject -class Win32_Process -computername $Com-
  puter -Filter “ProcessID=’$($service.processid)’” -ErrorAction Stop
               $hash.Add(“ProcessName”,$process.name)
               $hash.add(“VMSize”,$process.VirtualSize)
               $hash.Add(“PeakPageFile”,$process.PeakPageFileUsage)
               $hash.add(“ThreadCount”,$process.Threadcount)

               #create a custom object from the hash table
               $obj=New-Object -TypeName PSObject -Property $hash
               #add a type name to the custom object
               $obj.PSObject.TypeNames.Insert(0,’MOL.ServiceProcessInfo’)

               Write-Output $obj
```

```
            } #foreach service

            }
        Catch {
            #create an error message
            $msg="Failed to get service data from $computer. $($_.Exception.
   Message)"
            Write-Error $msg

            if ($LogErrors) {
                Write-Verbose "Logging errors to $errorlog"
                $computer | Out-File -FilePath $Errorlog -append
            }
        }

    } #foreach computer

} #process

End {
    Write-Verbose "Ending Get-ServiceInfo"
 }

}

Update-FormatData –prepend C:\CustomViewC.format.ps1xml

Get-ServiceInfo -ComputerName "localhost"
Get-ServiceInfo -ComputerName "localhost" | format-list
```

Chapter
13

Lab Answers

Script and Manifest Modules

Note: Labs A, B, and C for Chapters 7 through 14 build upon what was accomplished in previous chapters. If you haven't finished a lab from a previous chapter, please do so. Then check your results with sample solutions on More-Lunches.com before proceeding to the next lab in the sequence.

In this chapter you are going to assemble a module called PSHTools, from the functions and custom views that you've been working on for the last several chapters. Create a folder in the user module directory, called PSHTools. Put all of the files you will be creating in the labs into this folder.

Lab A

Create a single ps1xml file that contains all of the view definitions from the 3 existing format files. Call the file PSHTools.format.ps1xml. You'll need to be careful. Each view is defined by the <View></View> tags. These tags, and everything in between should go between the <ViewDefinition></ViewDefinition> tags.

Here is a sample solution:

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
    <ViewDefinitions>
        <View>
            <Name>MOL.SystemInfo</Name>
            <ViewSelectedBy>
                <TypeName>MOL.ComputerSystemInfo</TypeName>
            </ViewSelectedBy>
            <ListControl>
              <ListEntries>
                <ListEntry>
                    <ListItems>
                        <ListItem>
                            <PropertyName>ComputerName</PropertyName>
                        </ListItem>
                        <ListItem>
                            <PropertyName>Workgroup</PropertyName>
```

```
                    </ListItem>
                    <ListItem>
                        <PropertyName>AdminPassword</PropertyName>
                    </ListItem>
                    <ListItem>
                        <Propertyname>Model</Propertyname>
                    </ListItem>
                    <ListItem>
                        <Propertyname>Manufacturer</Propertyname>
                    </ListItem>
                    <ListItem>
                        <Propertyname>SerialNumber</Propertyname>
                        <Label>BIOSSerialNumber</Label>
                    </ListItem>
                    <ListItem>
                        <Propertyname>Version</Propertyname>
                        <Label>OSVersion</Label>
                    </ListItem>
                    <ListItem>
                        <Propertyname>ServicePackMajorVersion</Property-
name>
                        <Label>SPVersion</Label>
                    </ListItem>
                </ListItems>
            </ListEntry>
           </ListEntries>
        </ListControl>
    </View>
            <View>
        <Name>MOL.SystemInfo</Name>
        <ViewSelectedBy>
            <TypeName>MOL.DiskInfo</TypeName>
        </ViewSelectedBy>
        <TableControl>
            <TableHeaders>
                <TableColumnHeader>
                    <Width>18</Width>
                </TableColumnHeader>
                <TableColumnHeader/>
                <TableColumnHeader>
                    <Label>FreeSpace(GB)</Label>
                    <Width>15</Width>
                </TableColumnHeader>
                <TableColumnHeader>
                    <Label>Size(GB)</Label>
```

```
                <Width>10</Width>
            </TableColumnHeader>
        </TableHeaders>
        <TableRowEntries>
            <TableRowEntry>
                <TableColumnItems>
                    <TableColumnItem>
                        <PropertyName>ComputerName</PropertyName>
                    </TableColumnItem>
                    <TableColumnItem>
                        <PropertyName>Drive</PropertyName>
                    </TableColumnItem>
                    <TableColumnItem>
                        <PropertyName>FreeSpace</PropertyName>
                    </TableColumnItem>
                    <TableColumnItem>
                        <Propertyname>Size</Propertyname>
                    </TableColumnItem>
                </TableColumnItems>
            </TableRowEntry>
         </TableRowEntries>
    </TableControl>
</View>
<View>
    <Name>MOL.SystemInfo</Name>
    <ViewSelectedBy>
        <TypeName>MOL.ServiceProcessInfo</TypeName>
    </ViewSelectedBy>
    <TableControl>
        <TableHeaders>
            <TableColumnHeader>
                <Width>14</Width>
            </TableColumnHeader>
            <TableColumnHeader>
                <Label>Service</Label>
                <Width>13</Width>
            </TableColumnHeader>
            <TableColumnHeader>
                <Width>18</Width>
            </TableColumnHeader>
            <TableColumnHeader>
                <Width>17</Width>
            </TableColumnHeader>
            <TableColumnHeader>
                <Label>VM</Label>
```

```
            <Width>14</Width>
        </TableColumnHeader>
    </TableHeaders>
    <TableRowEntries>
        <TableRowEntry>
            <TableColumnItems>
                <TableColumnItem>
                    <PropertyName>ComputerName</PropertyName>
                </TableColumnItem>
                <TableColumnItem>
                    <PropertyName>Name</PropertyName>
                </TableColumnItem>
                <TableColumnItem>
                    <PropertyName>Displayname</PropertyName>
                </TableColumnItem>
                <TableColumnItem>
                    <Propertyname>ProcessName</Propertyname>
                </TableColumnItem>
                <TableColumnItem>
                    <Propertyname>VMSize</Propertyname>
                </TableColumnItem>
            </TableColumnItems>
        </TableRowEntry>
     </TableRowEntries>
</TableControl>
</View>
 <View>
<Name>MOL.SystemInfo</Name>
<ViewSelectedBy>
<TypeName>MOL.ServiceProcessInfo</TypeName>
</ViewSelectedBy>
 <ListControl>
  <ListEntries>
    <ListEntry>
        <ListItems>
            <ListItem>
                <PropertyName>ComputerName</PropertyName>
            </ListItem>
            <ListItem>
                <PropertyName>Name</PropertyName>
                <Label>Service</Label>
            </ListItem>
            <ListItem>
                <PropertyName>Displayname</PropertyName>
            </ListItem>
```

```
                        <ListItem>
                            <Propertyname>ProcessName</Propertyname>
                        </ListItem>
                        <ListItem>
                            <Propertyname>VMSize</Propertyname>
                        </ListItem>
                        <ListItem>
                            <Propertyname>ThreadCount</Propertyname>
                        </ListItem>
                        <ListItem>
                            <Propertyname>PeakPageFile</Propertyname>
                        </ListItem>
                    </ListItems>
                </ListEntry>
               </ListEntries>
            </ListControl>
        </View>
    </ViewDefinitions>
</Configuration>
```

Lab B

Create a single module file that contains the functions from the Labs A, B and C in Chapter 12, which should be the most current version. Export all functions in the module. Be careful to copy the function only. In your module file, also define aliases for your functions and export them as well.

Here is a sample solution:

```
#The PSHTools module file

Function Get-ComputerData {

<#
.SYNOPSIS
Get computer related data

.DESCRIPTION
This command will query a remote computer and return a custom object with sys-
   tem information pulled from WMI. Depending on the computer some information
   may not be available.

.PARAMETER Computername
The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.
```

```
.PARAMETER ErrorLog
Specify a path to a file to log errors. The default is C:\Errors.txt

.EXAMPLE
PS C:\> Get-ComputerData Server01
Run the command and query Server01.

.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-ComputerData -Errorlog c:\logs\
   errors.txt

This expression will go through a list of computernames and pipe each name to
   the command. Computernames that can't be accessed will be written to the log
   file.

#>

[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName,
 [string]$ErrorLog="C:\Errors.txt"
 )

 Begin {
    Write-Verbose "Starting Get-Computerdata"
 }

Process {
    foreach ($computer in $computerName) {
        Write-Verbose "Getting data from $computer"
        Try {
            Write-Verbose "Win32_Computersystem"
            $cs = Get-WmiObject -Class Win32_Computersystem -ComputerName $Com-
   puter -ErrorAction Stop

            #decode the admin password status
            Switch ($cs.AdminPasswordStatus) {
                1 { $aps="Disabled" }
                2 { $aps="Enabled" }
                3 { $aps="NA" }
                4 { $aps="Unknown" }
            }
```

```
        #Define a hashtable to be used for property names and values
        $hash=@{
            Computername=$cs.Name
            Workgroup=$cs.WorkGroup
            AdminPassword=$aps
            Model=$cs.Model
            Manufacturer=$cs.Manufacturer
        }

    } #Try

    Catch {

        #create an error message
        $msg="Failed getting system information from $computer. $($_.Excep-
tion.Message)"
        Write-Error $msg

        Write-Verbose "Logging errors to $errorlog"
        $computer | Out-File -FilePath $Errorlog -append

        } #Catch

    #if there were no errors then $hash will exist and we can continue and
assume
    #all other WMI queries will work without error
    If ($hash) {
        Write-Verbose "Win32_Bios"
        $bios = Get-WmiObject -Class Win32_Bios -ComputerName $Computer
        $hash.Add("SerialNumber",$bios.SerialNumber)

        Write-Verbose "Win32_OperatingSystem"
        $os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName
$Computer
        $hash.Add("Version",$os.Version)
        $hash.Add("ServicePackMajorVersion",$os.ServicePackMajorVersion)

        #create a custom object from the hash table
        $obj=New-Object -TypeName PSObject -Property $hash
        #add a type name to the custom object
        $obj.PSObject.TypeNames.Insert(0,'MOL.ComputerSystemInfo')

        Write-Output $obj
        #remove $hash so it isn't accidentally re-used by a computer that
causes
        #an error
```

```
            Remove-Variable -name hash
        } #if $hash
    } #foreach
} #process

 End {
    Write-Verbose “Ending Get-Computerdata”
 }
}

Function Get-VolumeInfo {

<#
.SYNOPSIS
Get information about fixed volumes

.DESCRIPTION
This command will query a remote computer and return information about fixed vol-
   umes. The function will ignore network, optical and other removable drives.

.PARAMETER Computername
The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.

.PARAMETER ErrorLog
Specify a path to a file to log errors. The default is C:\Errors.txt

.EXAMPLE
PS C:\> Get-VolumeInfo Server01
Run the command and query Server01.

.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-VolumeInfo -errorlog c:\logs\
   errors.txt

This expression will go through a list of computernames and pipe each name to
   the command. Computernames that can’t be accessed will be written to the log
   file.

#>
[cmdletbinding()]

 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName,
```

```
 [string]$ErrorLog="C:\Errors.txt",
  [switch]$LogErrors
 )

Begin {
    Write-Verbose "Starting Get-VolumeInfo"
 }

Process {
    foreach ($computer in $computerName) {
        Write-Verbose "Getting data from $computer"
        Try {
            $data = Get-WmiObject -Class Win32_Volume -computername $Computer
   -Filter "DriveType=3" -ErrorAction Stop

            Foreach ($drive in $data) {
                Write-Verbose "Processing volume $($drive.name)"
                #format size and freespace
                $Size="{0:N2}" -f ($drive.capacity/1GB)
                $Freespace="{0:N2}" -f ($drive.Freespace/1GB)

                #Define a hashtable to be used for property names and values
                $hash=@{
                    Computername=$drive.SystemName
                    Drive=$drive.Name
                    FreeSpace=$Freespace
                    Size=$Size
                }

                #create a custom object from the hash table
                $obj=New-Object -TypeName PSObject -Property $hash
                #Add a type name to the object
                $obj.PSObject.TypeNames.Insert(0,'MOL.DiskInfo')

                Write-Output $obj

            } #foreach

            #clear $data for next computer
            Remove-Variable -Name data

        } #Try

        Catch {
            #create an error message
```

```
            $msg=”Failed to get volume information from $computer. $($_.Excep-
   tion.Message)”
            Write-Error $msg

            Write-Verbose “Logging errors to $errorlog”
            $computer | Out-File -FilePath $Errorlog -append
        }
    } #foreach computer
} #Process

 End {
    Write-Verbose “Ending Get-VolumeInfo”
 }
}
Function Get-ServiceInfo {

<#
.SYNOPSIS
Get service information

.DESCRIPTION
This command will query a remote computer for running services and write a cus-
   tom object to the pipeline that includes service details as well as a few
   key properties from the associated process. You must run this command with
   credentials that have admin rights on any remote computers.

.PARAMETER Computername
The name of a computer to query. The account you use to run this function
   should have admin rights on that computer.

.PARAMETER ErrorLog
Specify a path to a file to log errors. The default is C:\Errors.txt

.PARAMETER LogErrors
If specified, computer names that can’t be accessed will be logged
to the file specified by -Errorlog.

.EXAMPLE
PS C:\> Get-ServiceInfo Server01

Run the command and query Server01.

.EXAMPLE
PS C:\> get-content c:\work\computers.txt | Get-ServiceInfo -logerrors

This expression will go through a list of computernames and pipe each name to
   the command. Computernames that can’t be accessed will be written to
```

```
the log file.

#>

[cmdletbinding()]
 param(
 [Parameter(Position=0,ValueFromPipeline=$True)]
 [ValidateNotNullorEmpty()]
 [string[]]$ComputerName,
 [string]$ErrorLog="C:\Errors.txt",
 [switch]$LogErrors
 )

 Begin {
    Write-Verbose "Starting Get-ServiceInfo"

    #if -LogErrors and error log exists, delete it.
    if ( (Test-Path -path $errorLog) -AND $LogErrors) {
        Write-Verbose "Removing $errorlog"
        Remove-Item $errorlog
    }
 }

 Process {

    foreach ($computer in $computerName) {
        Write-Verbose "Getting services from $computer"

        Try {
            $data = Get-WmiObject -Class Win32_Service -computername $Computer
   -Filter "State='Running'" -ErrorAction Stop

            foreach ($service in $data) {
                Write-Verbose "Processing service $($service.name)"
                $hash=@{
                Computername=$data[0].Systemname
                Name=$service.name
                Displayname=$service.DisplayName
                }

                #get the associated process
                Write-Verbose "Getting process for $($service.name)"
                $process=Get-WMIObject -class Win32_Process -computername $Com-
   puter -Filter "ProcessID='$($service.processid)'" -ErrorAction Stop
                $hash.Add("ProcessName",$process.name)
```

```
                $hash.add("VMSize",$process.VirtualSize)
                $hash.Add("PeakPageFile",$process.PeakPageFileUsage)
                $hash.add("ThreadCount",$process.Threadcount)

                #create a custom object from the hash table
                $obj=New-Object -TypeName PSObject -Property $hash
                #add a type name to the custom object
                $obj.PSObject.TypeNames.Insert(0,'MOL.ServiceProcessInfo')

                Write-Output $obj

            } #foreach service

            }
        Catch {
            #create an error message
            $msg="Failed to get service data from $computer. $($_.Exception.
   Message)"
            Write-Error $msg

            if ($LogErrors) {
                Write-Verbose "Logging errors to $errorlog"
                $computer | Out-File -FilePath $Errorlog -append
            }
        }

    } #foreach computer

} #process

End {
    Write-Verbose "Ending Get-ServiceInfo"
 }

}
#Define some aliases for the functions
New-Alias -Name gcd -Value Get-ComputerData
New-Alias -Name gvi -Value Get-VolumeInfo
New-Alias -Name gsi -Value Get-ServiceInfo

#Export the functions and aliases
Export-ModuleMember -Function * -Alias *
```

Lab C

Create a module manifest for the PSHTools that loads the module and custom format files. Test the module following these steps:

1. Import the module
2. Use Get-Command to view the module commands
3. Run help for each of your aliases
4. Run each command alias using localhost as the computername and verify formatting
5. Remove the module
6. Are the commands and variables gone?

Here is a sample manifest:

```
#
# Module manifest for module 'PSHTools'
#
# Generated by: Don Jones & Jeff Hicks
#


@{

# Script module or binary module file associated with this manifest.
RootModule = '.\PSHTools.psm1'

# Version number of this module.
ModuleVersion = '1.0'

# ID used to uniquely identify this module
GUID = '67afb568-1807-418e-af35-a296a43b6002'

# Author of this module
Author = 'Don Jones & Jeff Hicks'

# Company or vendor of this module
CompanyName = 'Month ofLunches'

# Copyright statement for this module
Copyright = '(c)2012 Don Jones and Jeffery Hicks'

# Description of the functionality provided by this module
Description = 'Chapter 13 Module for Month of Lunches'
```

```
# Minimum version of the Windows PowerShell engine required by this module
PowerShellVersion = '3.0'

# Name of the Windows PowerShell host required by this module
# PowerShellHostName = ''

# Minimum version of the Windows PowerShell host required by this module
# PowerShellHostVersion = ''

# Minimum version of the .NET Framework required by this module
# DotNetFrameworkVersion = ''

# Minimum version of the common language runtime (CLR) required by this module
# CLRVersion = ''

# Processor architecture (None, X86, Amd64) required by this module
# ProcessorArchitecture = ''

# Modules that must be imported into the global environment prior to importing
   this module
# RequiredModules = @()

# Assemblies that must be loaded prior to importing this module
# RequiredAssemblies = @()

# Script files (.ps1) that are run in the caller's environment prior to import-
   ing this module.
# ScriptsToProcess = @()

# Type files (.ps1xml) to be loaded when importing this module
# TypesToProcess = @()

# Format files (.ps1xml) to be loaded when importing this module
FormatsToProcess = '.\PSHTools.format.ps1xml'

# Modules to import as nested modules of the module specified in RootModule/
   ModuleToProcess
# NestedModules = @()

# Functions to export from this module
FunctionsToExport = '*'

# Cmdlets to export from this module
CmdletsToExport = '*'

# Variables to export from this module
```

```
VariablesToExport = '*'

# Aliases to export from this module
AliasesToExport = '*'

# List of all modules packaged with this module.
# ModuleList = @()

# List of all files packaged with this module
# FileList = @()

# Private data to pass to the module specified in RootModule/ModuleToProcess
# PrivateData = ''

# HelpInfo URI of this module
# HelpInfoURI = ''

# Default prefix for commands exported from this module. Override the default
   prefix using Import-Module -Prefix.
# DefaultCommandPrefix = ''

}
```

Chapter 16

Lab Answers

Making Tools that Make Changes

In WMI, the Win32_OperatingSystem class has a method called Win32Shutdown. It accepts a single input argument, which is a number that determines if the method shuts down, powers down, reboots, and logs off the computer.

Write a function called Set-ComputerState. Have it accept one or more computer names on a –ComputerName parameter. Also provide an –Action parameter, which accepts only the values LogOff, Restart, ShutDown, or PowerOff. Finally, provide a –Force switch parameter (switch parameters do not accept a value; they're either specified or not).

When the function runs, query Win32_OperatingSystem from each specified computer. Don't worry about error handling at this point – assume each specified computer will be available. Be sure to implement support for the –WhatIf and –Confirm parameters, as outlined in this chapter. Based upon the –Action specified, execute the Win32Shutdown method with one of the following values:

- LogOff – 0
- ShutDown – 1
- Restart – 2
- PowerOff – 8

If the –Force parameter is specified, add 4 to those values. So, if the command was Set-ComputerState –computername localhost –Action LogOff –Force, then the value would be 4 (zero for LogOff, plus 4 for Force). The execution of Win32Shutdown is what should be wrapped in the implementing If block for –WhatIf and –Confirm support.

Here is a sample solution:

```
Function Set-Computerstate {

[cmdletbinding(SupportsShouldProcess=$True,ConfirmImpact="High")]

Param (
[Parameter(Position=0,Mandatory=$True,HelpMessage="Enter a computername")]
[ValidateNotNullorEmpty()]
[string[]]$Computername,
[Parameter(Mandatory=$True,HelpMessage="Enter an action state")]
[ValidateSet("LogOff","Shutdown","Restart","PowerOff")]
```

```
[string]$Action,
[Switch]$Force

)
Begin {
    Write-Verbose “Starting Set-Computerstate”

    #set the state value
    Switch ($Action) {
    “LogOff”   { $Flag=0}
    “ShutDown” { $Flag=1}
    “Restart”  { $Flag=2}
    “PowerOff” { $Flag=8}
    }
    if ($Force) {
        Write-Verbose “Force enabled”
        $Flag+=4
    }
} #Begin

Process {
    Foreach ($computer in $Computername) {
        Write-Verbose “Processing $computer”
        $os=Get-WmiObject -Class Win32_OperatingSystem -ComputerName $Computer

        if ($PSCmdlet.ShouldProcess($computer)) {
            Write-Verbose “Passing flag $flag”
            $os.Win32Shutdown($flag)
        }

    } #foreach

} #Process

End {
    Write-Verbose “Ending Set-Computerstate”

} #end


} #close function

Set-Computerstate localhost -action LogOff -WhatIf -Verbose
```

Chapter 17

Lab Answers

Creating a Custom Type Extension

Revisit the advanced function that you wrote for Lab A in Chapters 6 through 14 of this book. Create a custom type extension for the object output by that function. Your type extension should be a ScriptMethod named CanPing(), as outlined in this chapter. Save the type extension file as PSHTools.ps1xml. Modify the PSHTools module manifest to load PSHTools.ps1xml, and then test your revised module to make sure the CanPing() method works.

Here is a sample ps1xml file:

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
  <Type>
    <Name>MOL.ComputerSystemInfo</Name>
    <Members>
        <ScriptMethod>
        <Name>CanPing</Name>
        <Script>
        Test-Connection -ComputerName $this.ComputerName -Quiet
        </Script>
    </ScriptMethod>
    </Members>
  </Type>
</Types>
```

Here is a sample ps1xml file:

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
  <Type>
    <Name>MOL.ComputerSystemInfo</Name>
    <Members>
        <ScriptMethod>
        <Name>CanPing</Name>
        <Script>
        Test-Connection -ComputerName $this.ComputerName -Quiet
        </Script>
    </ScriptMethod>
    </Members>
```

```
  </Type>
</Types>
```

Here is what the relevant part of the revised manifest might look like:

```
# Type files (.ps1xml) to be loaded when importing this module
TypesToProcess = '.\PSHTools.ps1xml'

# Format files (.ps1xml) to be loaded when importing this module
FormatsToProcess = '.\PSHTools.format.ps1xml'
```

Chapter
19

Lab Answers

Troubleshooting Pipeline Input

Create a text file named C:\Computers.csv. In it, place the following content:

```
ComputerName
LOCALHOST
NOTONLINE
```

Be sure there are no extra blank lines at the end of the file. Then, consider the following command:

```
Import-CSV C:\Computers.txt | Invoke-Command –Script { Get-Service }
```

The help file for Invoke-Command indicates that its –ComputerName parameter accepts pipeline input ByValue. Therefore, our expectation is that the computer names in the CSV file will be fed to the –ComputerName parameter. But if you run the command, that isn't what happens. Troubleshoot this command using the techniques described in this chapter, and determine where the computer names from the CSV file are being bound.

Chapter 20

Lab Answers

Using Object Hierarchies for Complex Output

Create a new function in your existing PSHTools module. Name the new function Get-Computer-VolumeInfo. This function's output will include some information that your other functions already produce, but this particular function is going to combine them all into a single, hierarchical object.

This function should accept one or more computer names on a –ComputerName parameter. Don't worry about error handling at this time. The output of this function should be a custom object with the following properties:

- ComputerName
- OSVersion (Version from Win32_OperatingSystem)
- SPVersion (ServicePackMajorVersion from Win32_OperatingSystem)
- LocalDisks (all instances of Win32_LogicalDisk having a DriveType of 3)
- Services (all instances of Win32_Service)
- Processes (all instances of Win32_ProcessS)

The function will therefore be making at least four WMI queries to each specified computer.

```
Function Get-ComputerVolumeInfo {


[cmdletbinding()]


Param([parameter(Position=0,mandatory=$True,

HelpMessage="Please enter a computername")]#

[ValidateNotNullorEmpty()]

[string[]]$Computername

)
```

```
Process {

  Foreach ($computer in $Computername) {
   Write-Verbose "Processing $computer"
   $params=@{Computername=$Computer;class="Win32_OperatingSystem"}

   Write-Verbose "Getting data from $($params.class)"
   #splat the parameters to the cmdlet
   $os = Get-WmiObject @params

   $params.Class="Win32_Service"
   Write-Verbose "Getting data from $($params.class)"
   $services = Get-WmiObject @params

   $params.Class="Win32_Process"
   Write-Verbose "Getting data from $($params.class)"
   $procs = Get-WmiObject @params

   $params.Class="Win32_LogicalDisk"
   Write-Verbose "Getting data from $($params.class)"
```

```
        $params.Add("filter","drivetype=3")
        $disks = Get-WmiObject @params

        New-Object -TypeName PSObject -property @{
          Computername=$os.CSName
          Version=$os.version
          SPVersion=$os.servicepackMajorVersion
          Services=$services
          Processes=$procs
          Disks=$disks
        }

      } #foreach computer
}
}

Get-ComputerVolumeInfo localhost
```

Chapter
22

Lab Answers

Crossing the Line: Utilizing the .NET Framework

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.

```
Function Resolve-HostIPAddress {



[cmdletbinding()]

Param (

[Parameter(Position=0,Mandatory=$True,

HelpMessage="Enter the name of a host. An FQDN is preferred.")]

[ValidateNotNullorEmpty()]

[string]$Hostname

)



Write-Verbose "Starting Resolve-HostIPAddress"

Write-Verbose "Resolving $hostname to IP Address"
```

```
Try {
    $data=[system.net.dns]::GetHostEntry($hostname)
    #the host might have multiple IP addresses
    Write-Verbose "Found $(($data.addresslist | measure-object).Count) address list entries"
    $data.AddressList | Select -ExpandProperty IPAddressToString
}
Catch {
    Write-Warning "Failed to resolve host $hostname to an IP address"
}

Write-Verbose "Ending Resolve-HostIPAddress"
} #end function

Resolve-HostIPAddress www.morelunches.com -verbose
```

Chapter 23

Lab Answers

Creating a GUI Tool, Part 1: The GUI

In this lab you're going to start a project that you'll work with over the next few chapters, so you'll want to make sure you have a working solution before moving on. Developing a graphical PowerShell script is always easier if you have a working command-line script. We've already done that part for you in the following listing.

LISTING CH23-LABFUNCTION

You can either retype or download the script from MoreLunches.com.

The function takes a computer name as a parameter and gets services via WMI based on user-supplied filter criteria. The function writes a subset of data to the pipeline. From the command line it might be used like this:

```
Get-servicedata $env:computername -filter running | Out-GridView
```

Your task in this lab is to create the graphical form using PowerShell Studio. You should end up with something like the form shown in figure 23.7.

Make the Running radio button checked by default. You'll find it easier later if you put the radio buttons in a GroupBox control, plus it looks cooler. The script you're creating doesn't have to do anything for this lab except display this form.

ANSWER - see code listing from MoreLunches.com Chapter 23.

Chapter 24

Lab Answers

Creating a GUI Tool, Part 2: The Code

In this lab you're going to continue where you left off in chapter 23. If you didn't finish, please do so first or download the sample solution from MoreLunches.com. Now you need to wire up your form and put some actions behind the controls.

First, set the Computername text box so that it defaults to the actual local computer name. Don't use localhost.

TIP Look for the form's Load event function.

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.

TIP You can avoid errors if you set the default behavior to search for running services.

You can test your form by sending output to Out-String and then Write-Host. For example, in your form you could end up with a line like this:

```
<code to get data> | Out-String | Write-Host
```

In the next chapter you'll learn better ways to handle form output.

ANSWER - see code listing from MoreLunches.com Chapter 24

Chapter 25

Lab Answers

Creating a GUI Tool, Part 3: The Output

We'll keep things pretty simple for this lab. Using the PowerShell Studio lab project from chapter 24, add a RichTextBox control to display the results. Here are some things to remember:

- Configure the control to use a fixed-width font like Consolas or Courier New.
- The Text property must be a string, so explicitly format data as strings by using-Out-String.
- Use the control's Clear() method to reset it or clear out any existing results.

If you need to move things around on your form, that's okay. You can download a sample solution at MoreLunches.com.

ANSWER - see code listing from MoreLunches.com Chapter 25

Chapter
26

Lab Answers

Creating Proxy Functions

Create a proxy function for the Export-CSV cmdlet. Name the proxy function Export-TDF. Remove the –Delimiter parameter, and instead hardcode it to always use –Delimiter "`t" (that's a backtick, followed by the letter t, in double quotation marks).

Work with the proxy function in a script file. At the bottom of the file, after the closing } of the function, put the following to test the function:

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

Run the script to test the function, and verify that it creates a tab-delimited file named c:\services.tdf.

ANSWER - see code listing from MoreLunches.com Chapter 26

Chapter 27

Lab Answers

Setting Up Constrained Demoting Endpoints

Create a new, local user named TestMan on your computer. Be sure to assign a pass- word to the account. Don't place the user in any user groups other than the default Users group.

Then, create a constrained endpoint on your computer. Name the endpoint ConstrainTest. Design it to include only the SmbShare module and to make only the Get-SmbShare command visible (in addition to a small core set of cmdlets like Exit-PSSession, Select-Object, and so forth). After creating the session configura- tion, register the endpoint. Configure the endpoint to permit only TestMan to con- nect (with Read and Execute permissions), and configure it to run all commands as your local Administrator account. Be sure to provide the correct password for Administrator when you're prompted.

Use Enter-PSSession to connect to the constrained endpoint. When doing so, use the –Credential parameter to specify the TestMan account, and provide the proper password when prompted. En- sure that you can run Get-SmbShare but not any other command (such as Get-SmbShareAccess).

ANSWER - see code listing from MoreLunches.com Chapter 27