Lab Sample Solutions

Chapter 4 lab

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, for example, Running. The whole point of toolmaking is to remain flexible and reusable without having to constantly edit the script. You should already have a start based on the examples in this chapter.

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 that 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

This script is supposed to create some new PSDrives based on environmental variables like `%APPDATA%` and `%USERPROFILE%\DOCUMENTS`. But 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. But once the script ends, the drives are still removed. If the intent was to make them visible in the console, then the solution would be to use a -Scope value of Global.

Chapter 6 lab

In these labs, 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 tools 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 filename but defaulting to C:\Errors.txt. Also plan ahead to create a custom view so that your function always outputs a table, using the following column headers:

- ComputerName
- Workgroup
- AdminPassword (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 but 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 two decimal places). Only include volumes that represent fixed hard drives—don't include optical or network drives in the output. Keep in mind that any 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 filename 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. But 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 you're getting information from a variety of WMI sources, a good function name might be `Get-ComputerData`. You'll need a string parameter for the name, a string for the log file, and maybe a switch parameter indicating that you want to log data. The func-

tion will need to make several WMI queries, and then it can write a custom object to the pipeline. You can get the computer name from one of the WMI classes. You could use the `computername` parameter, but by using something from WMI you'll get the "official" computer name, which is better if you test with something like localhost.

Because the `AdminStatus` property value is an integer, you can use a `Switch` statement to define a variable with the interpretation as a string.

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

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

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

But eventually you'll want to be able to pipe computer names to it. Each computer name should produce a custom object.

Lab B

Because the command will get volume data information, a likely name would be `Get-VolumeInfo` or `Get-VolumeData`. As in Lab A, you'll need a string parameter for a computer name, as well as a parameter for the event log and a switch to indicate whether or not to log errors. A sample command might look like this:

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

Also as in Lab A, using a hash table with the new properties will make it easier to create and write a custom object to the pipeline. You'll also need to convert the size and free space by dividing the size in bytes by 1 GB. 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 computer name, error log name, and whether or not to log files. Because you need to get the process id of each service, you'll need to use WMI or CIM. The `Get-Service` cmdlet returns a service object, but it doesn't include the process id. Once you have the service object you 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 two WMI classes. For now, you'll include all the properties. Later you can create a custom view with only the desired, default properties.

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

Chapter 7 lab

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're 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`, and so on. 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's possible that some values may be empty.

Here's 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 two 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's 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 (replacing `<function_name>` with your actual function name, which would not include the angle brackets).

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

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

Here's 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 three previous labs and work on this single, standalone 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 that the last boot time property does not contain a human-readable date/time value; you'll need to use the class's `ConvertToDateTime()` method to convert that value to a normal-looking date/time. Test the function by adding `Get-SystemInfo -computerName localhost` to the end of the script.

You should get a result like this:

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

Here's 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

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 the `-ComputerName` parameter won't 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's 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 won't 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's 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

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
```

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

Modify this function to accept pipeline input for the `-ComputerName` parameter. Add verbose output that will display the name of each computer contacted. Ensure that the `-ComputerName` parameter won't 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    : 6.1.7601
```

Here's 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
     $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
        }
    }
}

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

Chapter 9 Lab

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's 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 system 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
     $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"
 }
}

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's 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
volumes. 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's 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 the following listing, add comment-based help.

Listing 9.1 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
     $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
        }
    }
}
```

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's 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 addresses. 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
     $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
        }
    }
}
```

```
help Get-SystemInfo
```

Chapter 10 lab

You're 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 shouldn't 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
```

Here's 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 system 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
     $Computer -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.
     $($_.Exception.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

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-Volu
   meInfo

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

Here's 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
volumes. 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.
     $($_.Exception.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],
     WriteErrorException
    + FullyQualifiedErrorId :
     Microsoft.PowerShell.Commands.WriteErrorException,Get-Serv
   iceInfo

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

Here's a sample 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.

.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
     $Computer -Filter "ProcessID='$($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 the following listing as a starting point.

Listing 10.1 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 addresses. 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
     $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
        }
    }
}
```

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's run, so that it starts out fresh each time. If the first WMI query fails, the function should output nothing for that computer and shouldn't 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],
     WriteErrorException
    + 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's 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 addresses. 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.LastBootupTime))
                           '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

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 aren't part of the actual function. How would you

debug them? Revise them into working solutions. Remember, you'll need to dot source the script each time you make a change. We recommend testing in the regular PowerShell console.

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

Listing 11.1 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  }
```

COMMENTARY

The first step is to clean up the formatting a bit to make it easier to read. We're also going to delete the backticks so that one-line commands show as a single line. Ideally, you're doing this in a scripting editor or the ISE, something that will show you line numbers. Here's our first pass:

```
01  Function Get-ServiceInfo {
02
03  [cmdletbinding()]
04  Param([string]$Computername)
05
06  $services=Get-WmiObject -Class Win32_Services -filter "state='Running'" -
     computername $computernam
07
08  Write-Host "Found ($services.count) on $computername" -ForegroundColor
     Green
09
10  $sevices | sort -Property startname,name  Select -property
     startname,name,startmode,computername
11
12  }
```

This is short enough that problems might jump out at you, but we're going to dot source the script and try to run the function:

```
S:\Toolmaking> get-serviceinfo client2
Get-WmiObject : Cannot validate argument on parameter 'ComputerName'. The
     argument is null or empty. Supply an
argument that is not null or empty and then try the command again.
At S:\Toolmaking\a.ps1:6 char:86
+ ...  -computername $computernam
+                    ~~~~~~~~~~~~
    + CategoryInfo          : InvalidData: (:) [Get-WmiObject],
     ParameterBindingValidationException
```

```
    + FullyQualifiedErrorId :
     ParameterArgumentValidationError,Microsoft.PowerShell.Commands.GetWmiObj
     ectCommand

Found (.count) on client2
Sort-Object : Cannot bind parameter because parameter 'Property' is specified
     more than once. To provide multiple
values to parameters that can accept multiple values, use the array syntax.
     For example, "-parameter
value1,value2,value3".
At S:\Toolmaking\a.ps1:10 char:50
+ $sevices | sort -Property startname,name  Select -property
     startname,name,startm ...
+                                                  ~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Sort-Object],
     ParameterBindingException
    + FullyQualifiedErrorId :
     ParameterAlreadyBound,Microsoft.PowerShell.Commands.SortObjectCommand
```

There are a number of issues. Let's take them in order. The first problem is related to the `Get-WmiObject` expression. It appears there's a problem with the computer name parameter on line 6.

```
Get-WmiObject : Cannot validate argument on parameter 'ComputerName'. The
     argument is null or empty. Supply an
argument that is not null or empty and then try the command again.
At S:\Toolmaking\a.ps1:6 char:86
+ ...  -computername $computernam
```

And sure enough you can see here and in the script that the variable is misspelled. It should be `$computername`. We'll make this one change and repeat the test.

```
S:\Toolmaking> get-serviceinfo client2
Get-WmiObject : Invalid query "select * from Win32_Services where
     state='Running"
At S:\Toolmaking\a.ps1:6 char:11
+ $services=Get-WmiObject -Class Win32_Services -filter "state='Running" -
     computer ...
+
     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-WmiObject],
     ManagementException
    + FullyQualifiedErrorId :
     GetWMIManagementException,Microsoft.PowerShell.Commands.GetWmiObjectComm
     and

Found (.count) on client2
Sort-Object : Cannot bind parameter because parameter 'Property' is specified
     more than once. To provide multiple
values to parameters that can accept multiple values, use the array syntax.
     For example, "-parameter
value1,value2,value3".
At S:\Toolmaking\a.ps1:10 char:50
+ $sevices | sort -Property startname,name  Select -property
     startname,name,startm ...
```

```
+                                                  ~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Sort-Object],
     ParameterBindingException
    + FullyQualifiedErrorId :
     ParameterAlreadyBound,Microsoft.PowerShell.Commands.SortObjectCommand
```

We at least confirmed that we fixed the first problem. But now we see a new problem. This is where we start backing off and trying parts of our command. In the ISE you can select the first part of the `Get-WmiObject` command and test it out. Or you can do it at the prompt.

```
S:\Toolmaking> Get-WmiObject -Class Win32_Services
Get-WmiObject : Invalid class "Win32_Services"
At line:1 char:1
+ Get-WmiObject -Class Win32_Services
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidType: (:) [Get-WmiObject],
     ManagementException
    + FullyQualifiedErrorId :
     GetWMIManagementException,Microsoft.PowerShell.Commands.GetWmiObjectComm
     and
```

Well, that's a problem. Now it's possible for a WMI class to exist only on a given machine based on an installed application or product. We're testing this on Windows 7. The first step is to research this problem and either verify that this is the right class name or correct it. In this situation, a quick Internet search for `Win32_Services` shows that this should be `Win32_Service`. Back to the script, make one change, and try again.

```
S:\Toolmaking> get-serviceinfo client2
Get-WmiObject : Invalid query "select * from Win32_Service where
     state='Running"
At S:\Toolmaking\a.ps1:6 char:11
+ $services=Get-WmiObject -Class Win32_Service -filter "state='Running" -
     computern ...
+
     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-WmiObject],
     ManagementException
    + FullyQualifiedErrorId :
     GetWMIManagementException,Microsoft.PowerShell.Commands.GetWmiObjectComm
     and

Found (.count) on client2
Sort-Object : Cannot bind parameter because parameter 'Property' is specified
     more than once. To provide multiple
values to parameters that can accept multiple values, use the array syntax.
     For example, "-parameter
value1,value2,value3".
At S:\Toolmaking\a.ps1:10 char:50
+ $sevices | sort -Property startname,name  Select -property
     startname,name,startm ...
+                                                  ~~~~~~~~~
```

```
+ CategoryInfo          : InvalidArgument: (:) [Sort-Object],
 ParameterBindingException
+ FullyQualifiedErrorId :
 ParameterAlreadyBound,Microsoft.PowerShell.Commands.SortObjectCommand
```

Looks like there's still a problem. The only part left to the command that we haven't looked at is the filter. One thing we might want to do is use the WBEMTest utility and verify our query outside of PowerShell. From the prompt, type `WBEMTEST` and press Enter. Click Connect and verify that you're connecting to the root\cimv2 namespace. Click Connect again to connect to the local machine. Click the Query button. The error message shows the query. Grab everything inside the ""

```
select * from Win32_Service where state='Running
```

and paste it into the query window. Click Apply and you'll get the invalid query message. You probably noticed already that the query is missing the closing quote. Modify the query in WBEMTest:

```
select * from Win32_Service where state='Running'
```

Now try again. Success! Now you know what to fix in the script. Rinse and repeat.

```
S:\Toolmaking> get-serviceinfo client2
Found (\\CLIENT2\root\cimv2:Win32_Service.Name="AeLookupSvc"
     \\CLIENT2\root\cimv2:Win32_Service.Name="AudioEndpointBuild
er" \\CLIENT2\root\cimv2:Win32_Service.Name="BFE"
     \\CLIENT2\root\cimv2:Win32_Service.Name="BITS" \\CLIENT2\root\cimv2:Wi
n32_Service.Name="Browser"
     \\CLIENT2\root\cimv2:Win32_Service.Name="CertPropSvc"
     \\CLIENT2\root\cimv2:Win32_Service.Name
="CryptSvc" \\CLIENT2\root\cimv2:Win32_Service.Name="DcomLaunch"
     \\CLIENT2\root\cimv2:Win32_Service.Name="Dhcp" \\CLIENT
...
```

Whoa!! That's not what we expected. There's no line number, but we can see it displayed in green and it starts with `Found`, so this must be the result from line 8:

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

The problem is the `($services.count)` subexpression. It's missing a `$`. Line 8 should probably look like this:

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

Once more from the top:

```
S:\Toolmaking> get-serviceinfo client2
Found 65 on client2
Sort-Object : Cannot bind parameter because parameter 'Property' is specified
     more than once. To provide multiple
values to parameters that can accept multiple values, use the array syntax.
     For example, "-parameter
value1,value2,value3".
At S:\Toolmaking\a.ps1:10 char:50
```

```
+ $sevices | sort -Property startname,name  Select -property
     startname,name,startm ...
+                                                    ~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Sort-Object],
     ParameterBindingException
    + FullyQualifiedErrorId :
     ParameterAlreadyBound,Microsoft.PowerShell.Commands.SortObjectCommand
```

Better. Now there's a problem on line 10.

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

PowerShell is complaining that we're using `-Property` more than once, which we are, as you can see from the boldfaced listing: once for `Sort-Object` and again for `Select-Object`. Hold the phone. The function is supposed to sort and then pipe to `Select-Object` but there's no pipe character!

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

We'll make that change and test again.

```
S:\Toolmaking> get-serviceinfo client2
Found 65 on client2
```

Interesting. No errors but also no results. There are 65 services, so we should have gotten something. Line 10 is supposed to take all the services, sort them, and then select a few properties. We know what line is problematic, so let's try a breakpoint

```
PS S:\Toolmaking> Set-PSBreakpoint -Script .\a.ps1 -Line 10
```

We'll also include a new line with `Write-Debug` so we can check our variable.

```
07  Write-Debug "got services"
```

Running the function automatically puts us in debug mode.

```
PS S:\Toolmaking> Get-ServiceInfo client2
Hit Variable breakpoint on 'S:\Toolmaking\a.ps1:$services' (Write access)

At S:\Toolmaking\a.ps1:6 char:1
+ $services=Get-WmiObject -Class Win32_Service -filter "state='Running'" -
     computer ...
+
     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ~~~~~~~~
PS S:\Toolmaking>>
```

At the nested prompt you can type `?` to get help.

```
PS S:\Toolmaking>> ?

 s, stepInto         Single step (step into functions, scripts, etc.)
 v, stepOver         Step to next statement (step over functions, scripts,
     etc.)
 o, stepOut          Step out of the current function, script, etc.

 c, continue         Continue operation
```

```
 q, quit              Stop operation and exit the debugger

 k, Get-PSCallStack  Display call stack

 l, list              List source code for the current script.
                      Use "list" to start from the current line, "list <m>"
                      to start from line <m>, and "list <m> <n>" to list <n>
                      lines starting from line <m>

 <enter>              Repeat last command if it was stepInto, stepOver or list

 ?, h                 displays this help message.
```

Or you can check variables.

```
PS S:\Toolmaking>> $services[0]

ExitCode  : 0
Name      : AeLookupSvc
ProcessId : 200
StartMode : Manual
State     : Running
Status    : OK
```

That variable looks okay. Press c to continue.

```
PS S:\Toolmaking>> c
Found 66 on client2
Hit Line breakpoint on 'S:\Toolmaking\a.ps1:10'

At S:\Toolmaking\a.ps1:10 char:1
+ $sevices | sort -Property startname,name | Select -property
     startname,name,start ...
+
     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ~~~~~~~~
PS S:\Toolmaking>>
```

Let's check the variable:

```
PS S:\Toolmaking>> $sevices
```

What?

```
PS S:\Toolmaking>> get-variable $sevices
Get-Variable : Cannot validate argument on parameter 'Name'. The argument is
     null. Supply a non-null argument and try
the command again.
At line:1 char:14
+ get-variable $sevices
+              ~~~~~~~~
    + CategoryInfo          : InvalidData: (:) [Get-Variable],
     ParameterBindingValidationException
    + FullyQualifiedErrorId :
     ParameterArgumentValidationError,Microsoft.PowerShell.Commands.GetVariab
     leCommand
```

We know `$services` exist so upon closer examination we find a typo. We'll enter `q` to quit and revise the script. We'll also remove the breakpoints.

```
PS S:\Toolmaking> Get-ServiceInfo client2
Found 65 on client2

startname                    name                      startmode
     computername
---------                    ----                      ---------
     ------------
LocalSystem                   AudioEndpointBuilder       Auto
LocalSystem                   BITS                       Manual
...
```

So close. No errors, but we're also not getting all of the expected properties, which most likely means we have the wrong name. Researching again or testing with WBEMTest shows that we should be using `Systemname`. There's nothing left in the script, so hopefully this is the last problem.

```
PS S:\Toolmaking> Get-ServiceInfo client2
Found 65 on client2

startname                   name                      startmode
     systemname
---------                   ----                      ---------
     ----------
LocalSystem                 AudioEndpointBuilder       Auto
     CLIENT2
LocalSystem                 BITS                      Manual
     CLIENT2
LocalSystem                 Browser                   Manual
     CLIENT2
LocalSystem                 CertPropSvc               Manual
     CLIENT2
...
```

Finally, success. As you discovered if you stuck with us, oftentimes fixing one problem reveals another. Take it one step and one bug at a time. Don't change a half-dozen parts of your code at once. Sure, you might fix one problem but unintentionally create two more.

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.

Listing 11.2 Buggy export function

```
01  Function Export-EventLogSource {
02
03  [cmdletbinding()]
04  Param (
05  [Parameter(Position=0,Mandatory=$True,Helpmessage="Enter a
     computername",ValueFromPipeline=$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 $Computer -
     ErrorAction Stop
27  if ($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  }
```

Commentary

This is a much more complicated example. Ideally you won't format your code so poorly. But you might find a script on the Internet that you want to try out, in a test environment of course. So learning how to reformat is a good skill. Here's our revised script with line numbers.

```
01  Function Export-EventLogSource {
02
03  [cmdletbinding()]
04
05  Param (
06  [Parameter(Position=0,Mandatory=$True,Helpmessage="Enter a
     computername",ValueFromPipeline=$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
     $computername"
31
32      Try {
33          Write-Host $computername.ToUpper -ForegroundColor Green
34          $logs=Get-EventLog -LogName $log -Newest $Newest -Computer
     $Computer -ErrorAction Stop
35          if ($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,Message | 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
     $Computername"
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  }
```

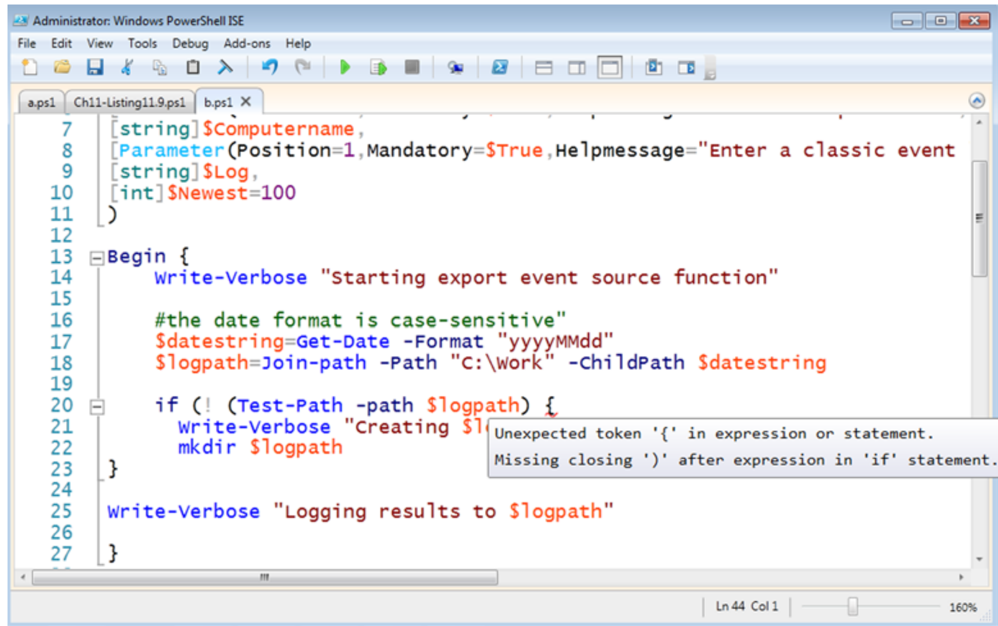

Figure 1

First off, you can save yourself some time by using a script editor or the ISE that includes syntax highlighting. If so, you can fix some problems pretty easily. Look at figure 1.

The ISE tells you exactly what the potential problem is. As you scroll through the rest of the file, you'll also see a message that the `Try` statement is missing its `Catch` or `Finally` script block.

As you can see in figure 2 there's a `Catch` block, so most likely we're missing a closing curly brace. Notice the – boxes to the left? These indicate collapsible sections. We can scroll up, collapsing sections as we go along and making sure that we're collapsing code between a set of curly braces. Other script editors might have other ways of matching braces. Figure 3 shows where we end up.

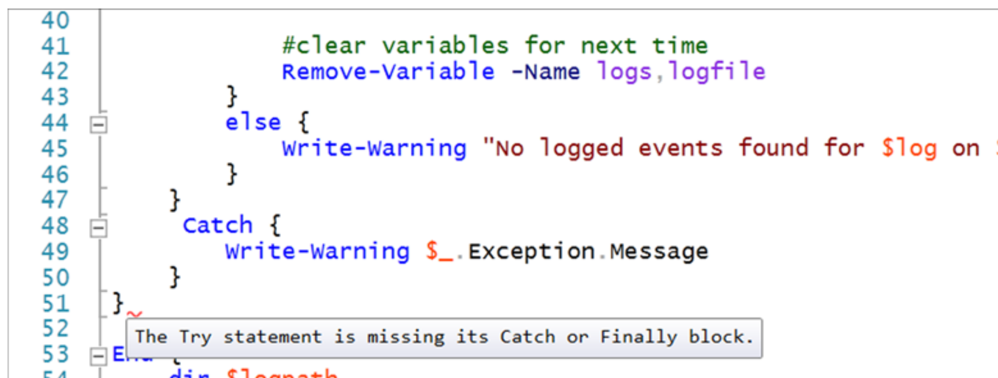

Figure 2

```
32 ⊟     Try {
33           Write-Host $computername.ToUpper -ForegroundColor Green
34           $logs=Get-EventLog -LogName $log -Newest $Newest -Computer $Compu
35 ⊞         if ($logs) {|...}
48 ⊞      Catch {...}
51     }
```

Figure 3

Interesting. What happened to the `Else` script block? It seems to have disappeared into the `If` script block, which is wrong. Let's expand it, as shown in figure 4.

The `If` script block must be using the brace after the `Else` script block, which leaves nothing for the `Try` block to use. The only thing we haven't looked at is the `foreach`. Perhaps it's missing a closing curly brace. We'll put one in at line 40 and see that the red squiggle will be gone. We can also start at the bottom and collapse each section, and it will look good. But all of this was the easy part. We still don't know what problems there are when we run it.

```
PS S:\Toolmaking> Export-EventLogSource -Computername client2 -Log System -
     Newest 10

    Directory: C:\Work

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         5/31/2012   2:01 PM            20120531
string ToUpper(), string ToUpper(cultureinfo culture)
WARNING: Cannot validate argument on parameter 'ComputerName'. The argument
     is null or empty. Supply an argument that
is not null or empty and then try the command again.
```

Well, it started out okay. The script is supposed to create a text log for each event source in the new directory and then display the files. We can see that the folder was created, so we got at least as far as line 22. Because the function has verbose messages, let's run it again and see if we can narrow down where the first problem begins.

```
PS S:\Toolmaking> Export-EventLogSource -Computername client2 -Log System -
     Newest 10 -verbose
VERBOSE: Starting export event source function
VERBOSE: Logging results to C:\Work\20120531
VERBOSE: Getting newest 10 System event log entries from client2
string ToUpper(), string ToUpper(cultureinfo culture)
WARNING: Cannot validate argument on parameter 'ComputerName'. The argument
     is null or empty. Supply an argument that
is not null or empty and then try the command again.
VERBOSE: Finished export event source function
```

```
32 ⊟     Try {
33           Write-Host $computername.ToUpper -ForegroundColor Green
34           $logs=Get-EventLog -LogName $log -Newest $Newest -Computer $Compu
35 ⊟         if ($logs) {
36               Write-Verbose "Sorting $($logs.count) entries"
37 ⊞             $log | sort Source | foreach {...}
44 ⊞         else {...}
47        }
48 ⊞      Catch {...}
51     }
```

Figure 4

Okay. Just after the verbose line that says what we're doing is some odd line about string `toUpper()`. Searching the script we find these lines:

```
30      Write-Verbose "Getting newest $newest $log event log entries from
     $computername"
31
32      Try {
33          Write-Host $computername.ToUpper -ForegroundColor Green
```

The script is trying to call the `ToUpper` method so that the computer name is in uppercase. But we need to include `()` when calling a method. Line 33 should be

```
33          Write-Host $computername.ToUpper() -ForegroundColor Green
```

Remember: one change at a time. Let's try again.

```
PS S:\Toolmaking> Export-EventLogSource -Computername client2 -Log System -
     Newest 10 -verbose
VERBOSE: Starting export event source function
VERBOSE: Logging results to C:\Work\20120531
VERBOSE: Getting newest 10 System event log entries from client2
CLIENT2
WARNING: Cannot validate argument on parameter 'ComputerName'. The argument
     is null or empty. Supply an argument that
is not null or empty and then try the command again.
VERBOSE: Finished export event source function
```

We corrected the first problem. Now there's a parameter problem. We know line 33 was the last thing successfully run, so we might need to take a look at line 34.

```
33          Write-Host $computername.ToUpper -ForegroundColor Green
34          $logs=Get-EventLog -LogName $log -Newest $Newest -Computer
     $Computer -ErrorAction Stop
```

Sure enough, there's a `-Computername` parameter, even though the function is using a shortened version. That's legal. And the value is `$Computer`. That's most likely the culprit. The error message says that it can't validate it because it's null or empty. Well, we just wrote the variable in uppercase so we know it has a value. Oh. We used `$computername` on line 33, and on line 34 we're using `$computer`. Those are two different variables. Most likely it should be `$computername`. We also could have used a breakpoint to step through the script.

```
PS S:\Toolmaking> Export-EventLogSource -Computername client2 -Log System -
     Newest 10 -verbose
VERBOSE: Starting export event source function
VERBOSE: Logging results to C:\Work\20120531
VERBOSE: Getting newest 10 System event log entries from client2
CLIENT2
VERBOSE: Sorting 10 entries

    Directory: C:\Work\20120531

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         5/31/2012   2:17 PM         18 client2-.txt
VERBOSE: Finished export event source function
```

No more errors, but we didn't get the expected results either. Each log is supposed to be sorted and content added to a text file.

```
38              $logfile=Join-Path -Path $logpath -ChildPath "$computername-
     $($_.Source).txt"
39              $_ | Format-List
     TimeWritten,MachineName,EventID,EntryType,Message | Out-File -FilePath
     $logfile -append
```

Line 38 defines the text file, which is supposed to be comprised of the computer name and the source. From the output we can see that the file is created in the right location but the name is wrong and it's missing the correct content:

```
PS S:\Toolmaking> cat C:\work\20120531\client2-.txt
System
```

One thing we might do is manually run the code that's getting the logs and verify that there are values for the `Source` property.

```
PS S:\Toolmaking> get-eventlog -LogName system -ComputerName client2 -newest
     10 | Select source

Source
------
Service Control Manager
Service Control Manager
Service Control Manager
Service Control Manager
Service Control Manager
EventLog
Service Control Manager
Service Control Manager
Service Control Manager
Service Control Manager
```

Okay. The command is good, and this means we should have had two files created. We can tell from our previous output that these lines of code are good:

```
34          $logs=Get-EventLog -LogName $log -Newest $Newest -Computer
     $Computer -ErrorAction Stop
35          if ($logs) {
36              Write-Verbose "Sorting $($logs.count) entries"
```

What we're not getting is a proper file created, so these lines are suspect:

```
37              $log | sort Source | foreach {
38              $logfile=Join-Path -Path $logpath -ChildPath "$computername-
     $($_.Source).txt"
39              $_ | Format-List
     TimeWritten,MachineName,EventID,EntryType,Message | Out-File -FilePath
     $logfile -append
```

It looks like line 37 is sorting all of the event logs and then doing something with each one in `ForEach`. But the script isn't piping `$logs`; it's piping `$log`. A simple typo but it makes sense. Because `$log` didn't exist, at least as an event log object, there was no source property, which is why our filename was incorrect. We'll fix it and retry.

```
PS S:\Toolmaking> Export-EventLogSource -Computername client2 -Log System -
     Newest 10 -verbose
VERBOSE: Starting export event source function
VERBOSE: Logging results to C:\Work\20120531
VERBOSE: Getting newest 10 System event log entries from client2
CLIENT2
VERBOSE: Sorting 10 entries

    Directory: C:\Work\20120531

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         5/31/2012   2:35 PM         18 client2-.txt
-a---         5/31/2012   2:44 PM        364 client2-EventLog.txt
-a---         5/31/2012   2:44 PM       3898 client2-Service Control
     Manager.txt
VERBOSE: Finished export event source function
```

That looks good. The function is supposed to write the time written, the computer name, the event id, the entry type, and the message to the file. Let's check one of the files:

```
PS S:\Toolmaking> get-content C:\work\20120531\client2-EventLog.txt

TimeWritten : 5/31/2012 12:00:05 PM
MachineName : CLIENT2.jdhlab.local
EventID     : 6013
EntryType   : Information
Message     : The system uptime is 90460 seconds.
```

This is what we expect, so the script has been cleaned up and debugged.

Chapter 12 lab

We bet you can guess what's 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 aren't 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 system 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
     $Computer -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.
     $($_.Exception.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 aren't 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
volumes. 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.
     $($_.Exception.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 computer name, 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           Background 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 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.

.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
     $Computer -Filter "ProcessID='$($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

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 three 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's 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</
     Propertyname>
                            <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 functions only. In your module file, also define aliases for your functions and export them as well.

Here's 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 system 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
     $Computer -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.
     $($_.Exception.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
volumes. 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.
     $($_.Exception.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 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.

.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
     $Computer -Filter "ProcessID='$($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` module 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 computer name and verify formatting.
5 Remove the module.
6 Are the commands and variables gone?

Here's 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
     importing 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

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 on 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` (`0` 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's 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

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's 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>
```

This 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

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.

SOLUTION

You can use Trace-Command to see what happens:

```
PS C:\> trace-command -name parameterbinding -pshost -expression {import-csv
     .\computers.csv | invoke-command {get-serv
ce}}
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Import-
     Csv]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args
     [Import-Csv]
DEBUG: ParameterBinding Information: 0 :     BIND arg [.\computers.csv] to
     parameter [Path]
DEBUG: ParameterBinding Information: 0 :         Binding collection parameter
     Path: argument type [String], parameter
type [System.String[]], collection type Array, element type [System.String],
     no coerceElementType
DEBUG: ParameterBinding Information: 0 :         Creating array with element
     type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 :         Argument type String is not
     IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 :         Adding scalar element of
     type String to array position 0
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
     metadata:
[System.Management.Automation.ValidateNotNullOrEmptyAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [System.String[]]
     to param [Path] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [Import-Csv]
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Invoke-
     Command]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args
     [Invoke-Command]
DEBUG: ParameterBinding Information: 0 :     BIND arg [get-service] to
     parameter [ScriptBlock]
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
     metadata:
[System.Management.Automation.ValidateNotNullAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [get-service] to
     param [ScriptBlock] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [Invoke-Command]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 :     BIND NAMED cmd line args [Get-
     Service]
DEBUG: ParameterBinding Information: 0 :     BIND POSITIONAL cmd line args
     [Get-Service]
```

```
DEBUG: ParameterBinding Information: 0 :     MANDATORY PARAMETER CHECK on
     cmdlet [Get-Service]
DEBUG: ParameterBinding Information: 0 :     CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
     [Invoke-Command]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
     [System.Management.Automation.PSCustomObject]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
     original values
DEBUG: ParameterBinding Information: 0 :     Parameter [InputObject] PIPELINE
     INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg
     [@{Computername=LOCALHOST}] to parameter [InputObject]
DEBUG: ParameterBinding Information: 0 :         BIND arg
     [@{Computername=LOCALHOST}] to param [InputObject] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [Invoke-Command]
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
     [Get-Service]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
     [System.Management.Automation.PSCustomObject]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
     original values
DEBUG: ParameterBinding Information: 0 :     Parameter [Name] PIPELINE INPUT
     ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg
     [@{Computername=LOCALHOST}] to parameter [Name]
DEBUG: ParameterBinding Information: 0 :         Binding collection parameter
     Name: argument type [PSObject], parameter
 type [System.String[]], collection type Array, element type [System.String],
     no coerceElementType
DEBUG: ParameterBinding Information: 0 :         Creating array with element
     type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 :         Argument type PSObject is
     not IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 :         BIND arg
     [@{Computername=LOCALHOST}] to param [Name] SKIPPED
DEBUG: ParameterBinding Information: 0 :     Parameter [Name] PIPELINE INPUT
     ValueFromPipelineByPropertyName NO
COERCION
DEBUG: ParameterBinding Information: 0 :     Parameter [ComputerName]
     PIPELINE INPUT ValueFromPipelineByPropertyName NO
 COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg [LOCALHOST] to
     parameter [ComputerName]
DEBUG: ParameterBinding Information: 0 :         Binding collection parameter
     ComputerName: argument type [String],
parameter type [System.String[]], collection type Array, element type
     [System.String], no coerceElementType
DEBUG: ParameterBinding Information: 0 :         Creating array with element
     type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 :         Argument type String is not
     IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 :         Adding scalar element of
     type String to array position 0
```

```
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
     metadata:
[System.Management.Automation.ValidateNotNullOrEmptyAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [System.String[]]
     to param [ComputerName] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 :     Parameter [Name] PIPELINE INPUT
     ValueFromPipeline WITH COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg
     [@{Computername=LOCALHOST}] to parameter [Name]
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
     [System.String[]]
DEBUG: ParameterBinding Information: 0 :             Trying to convert
     argument value from
System.Management.Automation.PSObject to System.String[]
DEBUG: ParameterBinding Information: 0 :             ENCODING arg into
     collection
DEBUG: ParameterBinding Information: 0 :             Binding collection
     parameter Name: argument type [PSObject],
parameter type [System.String[]], collection type Array, element type
     [System.String], coerceElementType
DEBUG: ParameterBinding Information: 0 :             Creating array with
     element type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 :             Argument type PSObject
     is not IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 :             COERCE arg to
     [System.String]
DEBUG: ParameterBinding Information: 0 :                 Trying to convert
     argument value from
System.Management.Automation.PSObject to System.String
DEBUG: ParameterBinding Information: 0 :                 CONVERT arg type to
     param type using
LanguagePrimitives.ConvertTo
DEBUG: ParameterBinding Information: 0 :                 CONVERT SUCCESSFUL
     using LanguagePrimitives.ConvertTo:
[@{Computername=LOCALHOST}]
DEBUG: ParameterBinding Information: 0 :             Adding scalar element of
     type String to array position 0
DEBUG: ParameterBinding Information: 0 :         BIND arg [System.String[]]
     to param [Name] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [Get-Service]
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
     [Out-Default]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
     [System.Management.Automation.ErrorRecord]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
     original values
DEBUG: ParameterBinding Information: 0 :     Parameter [InputObject] PIPELINE
     INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg [Cannot find any
     service with service name
'@{Computername=LOCALHOST}'.] to parameter [InputObject]
DEBUG: ParameterBinding Information: 0 :         BIND arg [Cannot find any
     service with service name
'@{Computername=LOCALHOST}'.] to param [InputObject] SUCCESSFUL
```

```
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [Out-Default]
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Out-
      LineOutput]
DEBUG: ParameterBinding Information: 0 :     BIND arg
      [Microsoft.PowerShell.Commands.Internal.Format.ConsoleLineOutput]
 to parameter [LineOutput]
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
      [System.Object]
DEBUG: ParameterBinding Information: 0 :             Parameter and arg types
      the same, no coercion is needed.
DEBUG: ParameterBinding Information: 0 :         BIND arg
[Microsoft.PowerShell.Commands.Internal.Format.ConsoleLineOutput] to param
      [LineOutput] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Out-
      LineOutput]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [Out-LineOutput]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
      [Out-LineOutput]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
      [System.Management.Automation.ErrorRecord]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
      original values
DEBUG: ParameterBinding Information: 0 :     Parameter [InputObject] PIPELINE
      INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg [Cannot find any
      service with service name
'@{Computername=LOCALHOST}'.] to parameter [InputObject]
DEBUG: ParameterBinding Information: 0 :         BIND arg [Cannot find any
      service with service name
'@{Computername=LOCALHOST}'.] to param [InputObject] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [out-lineoutput]
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Format-
      Default]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args
      [Format-Default]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [Format-Default]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
      [Format-Default]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
      [System.Management.Automation.ErrorRecord]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
      original values
DEBUG: ParameterBinding Information: 0 :     Parameter [InputObject] PIPELINE
      INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg [Cannot find any
      service with service name
'@{Computername=LOCALHOST}'.] to parameter [InputObject]
DEBUG: ParameterBinding Information: 0 :         BIND arg [Cannot find any
      service with service name
```

```
'@{Computername=LOCALHOST}'.] to param [InputObject] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [format-default]
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Set-
      StrictMode]
DEBUG: ParameterBinding Information: 0 :     BIND arg [1] to parameter
      [Version]
DEBUG: ParameterBinding Information: 0 :         Executing DATA GENERATION
      metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ArgumentToVersionTransfor
      mationAttribute]
DEBUG: ParameterBinding Information: 0 :             result returned from
      DATA GENERATION: 1.0
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
      [System.Version]
DEBUG: ParameterBinding Information: 0 :             Parameter and arg types
      the same, no coercion is needed.
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
      metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ValidateVersionAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [1.0] to param
      [Version] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Set-
      StrictMode]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [Set-StrictMode]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Set-
      StrictMode]
DEBUG: ParameterBinding Information: 0 :     BIND arg [1] to parameter
      [Version]
DEBUG: ParameterBinding Information: 0 :         Executing DATA GENERATION
      metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ArgumentToVersionTransfor
      mationAttribute]
DEBUG: ParameterBinding Information: 0 :             result returned from
      DATA GENERATION: 1.0
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
      [System.Version]
DEBUG: ParameterBinding Information: 0 :             Parameter and arg types
      the same, no coercion is needed.
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
      metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ValidateVersionAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [1.0] to param
      [Version] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Set-
      StrictMode]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [Set-StrictMode]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Set-
      StrictMode]
```

```
DEBUG: ParameterBinding Information: 0 :     BIND arg [1] to parameter
     [Version]
DEBUG: ParameterBinding Information: 0 :         Executing DATA GENERATION
     metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ArgumentToVersionTransfor
     mationAttribute]
DEBUG: ParameterBinding Information: 0 :             result returned from
     DATA GENERATION: 1.0
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
     [System.Version]
DEBUG: ParameterBinding Information: 0 :             Parameter and arg types
     the same, no coercion is needed.
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
     metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ValidateVersionAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [1.0] to param
     [Version] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Set-
     StrictMode]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [Set-StrictMode]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Set-
     StrictMode]
DEBUG: ParameterBinding Information: 0 :     BIND arg [1] to parameter
     [Version]
DEBUG: ParameterBinding Information: 0 :         Executing DATA GENERATION
     metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ArgumentToVersionTransfor
     mationAttribute]
DEBUG: ParameterBinding Information: 0 :             result returned from
     DATA GENERATION: 1.0
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
     [System.Version]
DEBUG: ParameterBinding Information: 0 :             Parameter and arg types
     the same, no coercion is needed.
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
     metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ValidateVersionAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [1.0] to param
     [Version] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Set-
     StrictMode]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [Set-StrictMode]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
get-service : Cannot find any service with service name
     '@{Computername=LOCALHOST}'.
At line:1 char:104
+ ... nvoke-command {get-service}}
+                      ~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound:
     (@{Computername=LOCALHOST}:String) [Get-Service], ServiceCommandExceptio
```

```
   n
    + FullyQualifiedErrorId :
     NoServiceFoundForGivenName,Microsoft.PowerShell.Commands.GetServiceComma
     nd

DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
     [Invoke-Command]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
     [System.Management.Automation.PSCustomObject]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
     original values
DEBUG: ParameterBinding Information: 0 :     Parameter [InputObject] PIPELINE
     INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg
     [@{Computername=NOTONLINE}] to parameter [InputObject]
DEBUG: ParameterBinding Information: 0 :         BIND arg
     [@{Computername=NOTONLINE}] to param [InputObject] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [Invoke-Command]
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
     [Get-Service]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
     [System.Management.Automation.PSCustomObject]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
     original values
DEBUG: ParameterBinding Information: 0 :     Parameter [Name] PIPELINE INPUT
     ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg
     [@{Computername=NOTONLINE}] to parameter [Name]
DEBUG: ParameterBinding Information: 0 :        Binding collection parameter
     Name: argument type [PSObject], parameter
 type [System.String[]], collection type Array, element type [System.String],
     no coerceElementType
DEBUG: ParameterBinding Information: 0 :         Creating array with element
     type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 :         Argument type PSObject is
     not IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 :         BIND arg
     [@{Computername=NOTONLINE}] to param [Name] SKIPPED
DEBUG: ParameterBinding Information: 0 :     Parameter [ComputerName]
     PIPELINE INPUT ValueFromPipelineByPropertyName NO
 COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg [NOTONLINE] to
     parameter [ComputerName]
DEBUG: ParameterBinding Information: 0 :        Binding collection parameter
     ComputerName: argument type [String],
parameter type [System.String[]], collection type Array, element type
     [System.String], no coerceElementType
DEBUG: ParameterBinding Information: 0 :         Creating array with element
     type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 :         Argument type String is not
     IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 :         Adding scalar element of
     type String to array position 0
```

```
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
     metadata:
[System.Management.Automation.ValidateNotNullOrEmptyAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [System.String[]]
     to param [ComputerName] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 :     Parameter [Name] PIPELINE INPUT
     ValueFromPipelineByPropertyName NO
COERCION
DEBUG: ParameterBinding Information: 0 :     Parameter [Name] PIPELINE INPUT
     ValueFromPipeline WITH COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg
     [@{Computername=NOTONLINE}] to parameter [Name]
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
     [System.String[]]
DEBUG: ParameterBinding Information: 0 :             Trying to convert
     argument value from
System.Management.Automation.PSObject to System.String[]
DEBUG: ParameterBinding Information: 0 :             ENCODING arg into
     collection
DEBUG: ParameterBinding Information: 0 :             Binding collection
     parameter Name: argument type [PSObject],
parameter type [System.String[]], collection type Array, element type
     [System.String], coerceElementType
DEBUG: ParameterBinding Information: 0 :             Creating array with
     element type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 :             Argument type PSObject
     is not IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 :             COERCE arg to
     [System.String]
DEBUG: ParameterBinding Information: 0 :                 Trying to convert
     argument value from
System.Management.Automation.PSObject to System.String
DEBUG: ParameterBinding Information: 0 :                 CONVERT arg type to
     param type using
LanguagePrimitives.ConvertTo
DEBUG: ParameterBinding Information: 0 :                 CONVERT SUCCESSFUL
     using LanguagePrimitives.ConvertTo:
[@{Computername=NOTONLINE}]
DEBUG: ParameterBinding Information: 0 :            Adding scalar element of
     type String to array position 0
DEBUG: ParameterBinding Information: 0 :         BIND arg [System.String[]]
     to param [Name] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [Get-Service]
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
     [Out-Default]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
     [System.Management.Automation.ErrorRecord]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
     original values
DEBUG: ParameterBinding Information: 0 :     Parameter [InputObject] PIPELINE
     INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg [Cannot find any
     service with service name
'@{Computername=NOTONLINE}'.] to parameter [InputObject]
```

```
DEBUG: ParameterBinding Information: 0 :         BIND arg [Cannot find any
     service with service name
'@{Computername=NOTONLINE}'.] to param [InputObject] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [Out-Default]
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
     [Out-LineOutput]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
     [System.Management.Automation.ErrorRecord]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
     original values
DEBUG: ParameterBinding Information: 0 :     Parameter [InputObject] PIPELINE
     INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg [Cannot find any
     service with service name
'@{Computername=NOTONLINE}'.] to parameter [InputObject]
DEBUG: ParameterBinding Information: 0 :         BIND arg [Cannot find any
     service with service name
'@{Computername=NOTONLINE}'.] to param [InputObject] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [out-lineoutput]
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
     [Format-Default]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
     [System.Management.Automation.ErrorRecord]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
     original values
DEBUG: ParameterBinding Information: 0 :     Parameter [InputObject] PIPELINE
     INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg [Cannot find any
     service with service name
'@{Computername=NOTONLINE}'.] to parameter [InputObject]
DEBUG: ParameterBinding Information: 0 :         BIND arg [Cannot find any
     service with service name
'@{Computername=NOTONLINE}'.] to param [InputObject] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
     [format-default]
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Set-
     StrictMode]
DEBUG: ParameterBinding Information: 0 :     BIND arg [1] to parameter
     [Version]
DEBUG: ParameterBinding Information: 0 :         Executing DATA GENERATION
     metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ArgumentToVersionTransfor
     mationAttribute]
DEBUG: ParameterBinding Information: 0 :             result returned from
     DATA GENERATION: 1.0
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
     [System.Version]
DEBUG: ParameterBinding Information: 0 :             Parameter and arg types
     the same, no coercion is needed.
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
     metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ValidateVersionAttribute]
```

```
DEBUG: ParameterBinding Information: 0 :         BIND arg [1.0] to param
      [Version] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Set-
      StrictMode]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [Set-StrictMode]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Set-
      StrictMode]
DEBUG: ParameterBinding Information: 0 :     BIND arg [1] to parameter
      [Version]
DEBUG: ParameterBinding Information: 0 :         Executing DATA GENERATION
      metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ArgumentToVersionTransfor
      mationAttribute]
DEBUG: ParameterBinding Information: 0 :             result returned from
      DATA GENERATION: 1.0
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
      [System.Version]
DEBUG: ParameterBinding Information: 0 :             Parameter and arg types
      the same, no coercion is needed.
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
      metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ValidateVersionAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [1.0] to param
      [Version] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Set-
      StrictMode]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [Set-StrictMode]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Set-
      StrictMode]
DEBUG: ParameterBinding Information: 0 :     BIND arg [1] to parameter
      [Version]
DEBUG: ParameterBinding Information: 0 :         Executing DATA GENERATION
      metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ArgumentToVersionTransfor
      mationAttribute]
DEBUG: ParameterBinding Information: 0 :             result returned from
      DATA GENERATION: 1.0
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
      [System.Version]
DEBUG: ParameterBinding Information: 0 :             Parameter and arg types
      the same, no coercion is needed.
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
      metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ValidateVersionAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [1.0] to param
      [Version] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Set-
      StrictMode]
```

```
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [Set-StrictMode]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Set-
      StrictMode]
DEBUG: ParameterBinding Information: 0 :     BIND arg [1] to parameter
      [Version]
DEBUG: ParameterBinding Information: 0 :         Executing DATA GENERATION
      metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ArgumentToVersionTransfor
      mationAttribute]
DEBUG: ParameterBinding Information: 0 :             result returned from
      DATA GENERATION: 1.0
DEBUG: ParameterBinding Information: 0 :         COERCE arg to
      [System.Version]
DEBUG: ParameterBinding Information: 0 :             Parameter and arg types
      the same, no coercion is needed.
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
      metadata:
[Microsoft.PowerShell.Commands.SetStrictModeCommand+ValidateVersionAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [1.0] to param
      [Version] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Set-
      StrictMode]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
      [Set-StrictMode]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
get-service : Cannot find any service with service name
      '@{Computername=NOTONLINE}'.
At line:1 char:104
+ ... nvoke-command {get-service}}
+                    ~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound:
      (@{Computername=NOTONLINE}:String) [Get-Service], ServiceCommandExceptio
   n
    + FullyQualifiedErrorId :
      NoServiceFoundForGivenName,Microsoft.PowerShell.Commands.GetServiceComma
      nd

DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
DEBUG: ParameterBinding Information: 0 :     CALLING EndProcessing
```

Chapter 20 lab

Create a new function in your existing `PSHTools` module. Name the new function `Get-ComputerVolumeInfo`. 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.

Here's one possible solution:

```
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

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.

Here's one possible solution:

```
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 26 lab

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.

Here's one possible solution with comments that explain what we did:

```
<#
First, we need to run these lines to create the metadata:

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

```
[System.Management.Automation.ProxyCommand]::Create($metadata) | Out-File
     ProxyExportCSV.ps1

#>

Function Export-TDF {

#we deleted the help link in cmdletbinding and added our own

<#
.Synopsis
Export to tab delimited file
.Description
This is a proxy command to Export-CSV which is hard coded to export
data to a tab-delimited file.
#>

[CmdletBinding(DefaultParameterSetName='Delimiter',
SupportsShouldProcess=$true,
ConfirmImpact='Medium'
)
]
param(
    [Parameter(Mandatory=$true, ValueFromPipeline=$true,
     ValueFromPipelineByPropertyName=$true)]
    [psobject]
    ${InputObject},

    [Parameter(Position=0)]
    [ValidateNotNullOrEmpty()]
    [string]
    ${Path},

    [Alias('PSPath')]
    [ValidateNotNullOrEmpty()]
    [string]
    ${LiteralPath},

    [switch]
    ${Force},

    [Alias('NoOverwrite')]
    [switch]
    ${NoClobber},

[ValidateSet('Unicode','UTF7','UTF8','ASCII','UTF32','BigEndianUnicode','Defa
     ult','OEM')]
    [string]
    ${Encoding},

    [switch]
    ${Append},

    #we deleted the Delimiter parameter that used to be here

    [Parameter(ParameterSetName='UseCulture')]
    [switch]
    ${UseCulture},

    [Alias('NTI')]
    [switch]
```

```
    ${NoTypeInformation})

begin
{
    try {
        $outBuffer = $null
        if ($PSBoundParameters.TryGetValue('OutBuffer', [ref]$outBuffer))
        {
            $PSBoundParameters['OutBuffer'] = 1
        }
        $wrappedCmd = $ExecutionContext.InvokeCommand.GetCommand('Export-
     Csv', [System.Management.Automation.CommandTypes]::Cmdlet)
        <#
        we added a hard coded reference to include the original -delimiter
     parameter
        with the tab character.
        #>
        $scriptCmd = {& $wrappedCmd @PSBoundParameters -delimiter "`t"}
        $steppablePipeline =
     $scriptCmd.GetSteppablePipeline($myInvocation.CommandOrigin)
        $steppablePipeline.Begin($PSCmdlet)
    } catch {
        throw
    }
}

process
{
    try {
        $steppablePipeline.Process($_)
    } catch {
        throw
    }
}

end
{
    try {
        $steppablePipeline.End()
    } catch {
        throw
    }
}
#We deleted the links for forwarded help
} #end function

#test it out
Get-Service | Export-TDF c:\services.tdf
```

Chapter 27 lab

Create a new, local user named TestMan on your computer. Be sure to assign a password 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 only include 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 configuration, register the endpoint. Configure the endpoint to permit only TestMan to connect (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. Ensure that you can run `Get-SmbShare` but not any other command (such as `Get-SmbShareAccess`).

Here's one possible solution.

First, create the session configuration file:

```
New-PSSessionConfigurationFile -Path C:\Scripts\ConstrainTest.pssc `
                               -Description 'Chapter 27 lab' `
                               -ExecutionPolicy Restricted `
                               -ModulesToImport SMBShare `
                               -PowerShellVersion 3.0 `
                               -VisibleFunctions 'Get-SMBShare'`
                               -SessionType RestrictedRemoteServer
```

Next, you need to register it:

```
Register-PSSessionConfiguration -Path C:\Scripts\ConstrainTest.pssc `
                                -Name ConstrainTest `
                                -ShowSecurityDescriptorUI `
                                -AccessMode Remote `
                                -RunAsCredential Administrator
```

To test, enter the session using alternate credentials:

```
PS C:\> enter-pssession -ComputerName localhost -ConfigurationName
     ConstrainTest -Credential testman
```

You can see what commands are available to you:

```
[quark]: PS>get-command

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Exit-PSSession
Function        Get-Command
Function        Get-FormatData
Function        Get-Help
Function        Get-SmbShare                                       SMBShare
Function        Measure-Object
Function        Out-Default
Function        Select-Object
```

Now try to run an authorized command:

```
[quark]: PS>get-process
The term 'get-process' is not recognized as the name of a cmdlet, function,
     script file, or
```

```
operable program. Check the spelling of the name, or if a path was included,
     verify that the path
is correct and try again.
    + CategoryInfo          : ObjectNotFound: (get-process:String) [],
     CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```