ScriptingAnswers.com

Windows PowerShell Fundamentals

LAB GUIDE

Lab 2-1
Navigating Your System

Task #1

Here’s your task:
Get a listing of all files and folders located in your C:\Program Files folder.

Here’s a tip:
The **Dir** command will do the job. Remember that it has a parameter which forces it to recurse subdirectories. Use the **Help** command to ask for help on **Dir** if necessary.

Task #2

Here’s your task:
Get a listing of all subkeys in the SOFTWARE key of the HKEY_CURRENT_USER registry hive.

Here’s a tip:
Remember that the registry is exposed as two “drives,” named HKLM: and HKCU:. Also remember that the colon is required at the end of a drive name, just as in Cmd.exe.

Task #3

Here’s your task:
Display the contents of the **systemroot** environment variable.

Here’s a tip:
Remember that the environment variable store is exposed as a drive. You can use the **Get-PSDrive** command to display a list of attached drives and their drive names.

Here’s a tip:
The environment drive is named ENV:.

Here’s a tip:
In Cmd.exe, you’d use the **Type** command to display the contents of a file. The items contained within any PSDrive, including ENV:, behave as files.

Task #4

Here’s your task:
Create a new registry key named SAPIENClasses, under the SOFTWARE key of the HKEY_CURRENT_USER registry hive. Do so by copying an existing registry key, rather than creating a new, blank key.

Here’s a tip:
Remember that registry keys “look like” directories, and that the two main registry hives are “mapped” as PSDrives.

Here’s a tip:
The HKEY_CURRENT_USER hive is mapped as the drive HKCU:

Here’s a tip:
Registry keys are the same thing as directories on the file system; what command would you use to copy a directory?

Here’s a tip:
Notice that the correct command only copies the key – not its contents. What would you need to add to also copy its contents?

Task #5

Here’s your task:
Remove the registry key named SAPIENClasses, under the SOFTWARE key of the HKEY_CURRENT_USER registry hive.

Here’s a tip:
Remember that registry keys “look like” directories, and that the two main registry hives are “mapped” as PSDrives.

Here’s a tip:
The HKEY_CURRENT_USER hive is mapped as the drive HKCU:

Lab 3-1
Finding and Using Cmdlets and Parameters

Task #1

Here's your task:
What cmdlet is referenced by the alias Gci?

Here's a tip:
Remember that you can always ask for help on an alias to see what cmdlet the alias "points" to.

Task #2

Here's your task:
Create a new alias, D, that displays a list of child items.

Here's a tip:
You should already know the cmdlet that you need to create an alias for, and you can ask for help on *Alias* to see what cmdlets are available to work with aliases.

Task #3

Here's your task:
In what snap-in is the cmdlet Get-WMIObject defined?

Here's a tip:
PowerShell uses the noun PSSnapIn to refer to snap-ins. You can ask for help with *snapin* to see what cmdlets are available for working with them.

Here's a tip:
Once you know a snap-in's name, you can use that name with a Get-Command parameter to get the cmdlets for a specific snap-in.

Task #4

Here's your task:
What cmdlets will allow you to start and stop the creation of a transcript that contains everything typed into the shell?

Here's a tip:
Remember, you've been shown several ways to discover new cmdlet names.

Task #5

Here's your task:
Use Windows to launch Windows Calculator (or type **Calc** in PowerShell). Now, what command would you type to stop, or kill, the Calc process from within PowerShell?

Here's a tip:
Remember, you've been shown several ways to discover new cmdlet names. Try asking for help on *Process* or using Get-Command with its –noun parameter. Or, if you know of a command from Cmd.exe that would do this, see if that command is defined as an alias within PowerShell.

Task #6

Here's your task:
Use Notepad to create a text file that contains at least thirty lines of text. Then, use PowerShell to display just the first ten lines.

Here's a tip:
What command would you use in Cmd.exe to display the contents of a text file? Perhaps that command is an alias in PowerShell.

Here's a tip:
Think about the noun "Content" in a cmdlet name.

Here's a tip:
Remember to ask for help on a cmdlet to see what it's capable of.

Task #7

Here's your task:
Display the current date only – not the time.

Here's a tip:
Keep in mind that any cmdlet which will retrieve something uses the Get verb.

Here's a tip:
Ask for help with a cmdlet to see what options it offers.

Task #8

Here's your task:
Display the most recent 20 events from the Security event log.

Here's a tip:
Keep in mind that any cmdlet which will retrieve something uses the Get verb.

Here's a tip:
This cmdlet probably uses a noun like EventLog or EventLogEntry or EventLogItem – try using these nouns with the –noun parameter of Get-Command.

Here's a tip:
Ask for help with a cmdlet to see what options it offers.

Lab 4-1
Working in the Pipeline

Task #1

Here's your task:
What properties are available for a Service object?

Here's a tip:
The Get-Service cmdlet returns a collection of service objects.

Here's a tip:
What cmdlet could you *pipe* the services to in order to *get* a list of the services' *members* (or properties)?

Task #2

Here's your task:
If the pipeline ends in Out-Default, which produces text on the screen, how do you think you could have text go into a file instead? For example, how could you send a list of processes into a file named c:\processes.txt?

Here's a tip:
If Out-Default goes to the default display—the screen—then perhaps another Out cmdlet sends output somewhere else.

Task #3

Here's your task:
How can you tell PowerShell to stop all services on your computer—but have the shell only list what it would do, rather than actually stopping them?

Here's a tip:
Remember, there's a parameter for most reconfiguration cmdlets that does this.

Task #4

Here's your task:
How could you tell PowerShell to stop all services on your computer, but have the shell actually ask you "yes or no" for each service, rather than doing them all automatically?

Here's a tip:
If one parameter is available to display what PowerShell would have done, perhaps another parameter exists that would have the shell verify or confirm the action on an object-by-object basis

Task #5

Here's your task:
Create a folder named C:\Test on your computer. Then, create a text file named C:\Files.txt. In that file, list three complete file paths and names—one per line—of files that already exist on your system.

Now, tell PowerShell to copy all of the files listed in C:\Files.txt into C:\Test, using a single command-line.

Here's a tip:
You should already know the commands for reading content from a text file and for copying items in the file system. You just need to combine them somehow.

The Copy-Item cmdlet will do the trick. Does it have a parameter that accepts a "source" filename? Can that parameter accept pipeline input? Perhaps asking for –full help would be useful.

Lab 5-1
Using Core Cmdlets

Task #1

Here's your task:
Do the objects returned by Get-Service have a StartMode property?

Here's a tip:
You can pipe an object to Get-Member to see a list of that object's properties.

Task #2

Here's your task:
What property of an event log entry contains the numeric event ID?

Here's a tip:
You can pipe an object to Get-Member to see a list of that object's properties.

Here's a tip:
Need to know what cmdlet returns event log entry objects? Ask **Help** for ***event*** to see what cmdlets are available.

Task #3

Here's your task:
Display a list of the 10 processes using the most handles.

Here's a tip:
You'll need to sort the processes on an appropriate property first.

Here's a tip:
You'll need to use a cmdlet to select just the "top 10" processes.

Task #4

Here's your task:
Display Running and Stopped services grouped together, with the total number of Running and Stopped services displayed.

Here's a tip:
There's a cmdlet that will group and count objects.

Task #5

Here's your task:
How many services are installed on your system?

Here's a tip:
You don't need to count – Get-Service returns a collection of services, and a cmdlet is available to measure the size of that collection (e.g., how many objects it contains).

Task #6

Here's your task:
What is the average number of handles in use by processes on your system? What is the largest number of handles in use by a process? The smallest?

Here's a tip:
You can find all of this information in a single command-line.

Here's a tip:
Remember that averages, minimums, and maximums are a form of measurement. Ask PowerShell for **Help** with *measure* if you need assistance.

Task #7

Here's your task:
Create a list of event log entries from the Security event log, including only the event ID and the time of the event.

Here's a tip:
A cmdlet is available that accepts objects, and returns only the selected properties of each object.

Lab 6-1
Using Security Features

Task #1

Here’s your task:
Reconfigure PowerShell so that local scripts will run without being signed.

Here’s a tip:
You’re changing (setting) the ExecutionPolicy.

Task #2

Here’s your task:
Use Notepad or PrimalScript to create a file named Test.ps1 in c:\test. Put the following into the file:

```
Write-Host "I ran"
```

In PowerShell, change to the c:\test directory, and run the script.

Here’s a tip:
You can’t run scripts just by typing their name.

Task #3

Here’s your task:
What cmdlet could you use to verify the signature on a signed script? What cmdlet could you use to apply a signature to a script (if you had a code-signing certificate installed)?

Here’s a tip:
Script signatures are called AuthenticodeSignatures

Here’s a tip:
Ask for help with *signature* to see what comes up.

Task #4

Here's your task:
Configure PowerShell so that, each time the shell loads, it changes to the C:\Test folder.

Here's a tip:
You'll need to make a special kind of script file in order for this to work.

Task #5

Here's your task:
Configure PowerShell so that, each time the shell loads, it prompts you for the local Administrator password and stores the completed credential in a variable named $cred.

Here's a tip:
You'll need to modify the file you created in the previous task.

Here's a tip:
Can't remember the cmdlet name? Ask for help with *credential* to see what's available.

Lab 7-1
Using WMI in PowerShell

Task #1

Here's your task:
What class do you think would represent a:

- Windows service?
- Logical disk drive?
- Network adapter's configuration settings?
- Running process?
- The Windows operating system itself

Here's a tip:
These classes all exist in the root\cimv2 namespace.

Here's a tip:
Get-WMIObject can *list* the classes from a given namespace, or you can use the documentation.

Task #2

Here's your task:
Retrieve all of the logical disks from your local computer.

Here's a tip:
Use the Get-WMIObject cmdlet. No need to specify a computer name if you're querying your local computer.

Task #3

Here's your task:
Retrieve all of the logical disks from your local computer, sorting them by their drive type. Try mapping a network drive or two, if possible, to create more drives in the list.

Here's a tip:
Remember that WMI objects, as returned by Get-WMIObject, can be used in the pipeline like any other object.

Task #4

Here's your task:
Use WMI to get a list of all services which are running.

Here's a tip:
The Win32_Service class has a property named State which should be "Running" for services which are running.

Here's a tip:
The Get-WMIObject cmdlet has a –filter parameter which can filter out objects you don't want. Look up help for information on using it. Start by getting *all* objects, and then try re-running the command with a filter applied.

Here's a tip:
The –filter parameter passes the filter criteria through to WMI, which does the filtering. So you'll use a WQL-style filter criteria, without a WHERE keyword.

Task #5

Here's your task:
Repeat Task #4, but this time only list the name and start mode of running services.

Here's a tip:
You can pipe the results of Get-WMIObject to another cmdlet to select the properties you want.

Task #6

Here's your task:
Use the –query parameter of Get-WMIObject to retrieve all properties of the Win32_LogicalDisk class, for local drives only.

Here's a tip:
The Win32_LogicalDisk class has a DriveType property which tells you what kind of drive each class instance actually is.

Here's a tip:
You need to write a WQL query with a WHERE clause; you can't use the –filter parameter in addition to the –query parameter (see Help for Get-WMIObject).

Task #7

Here’s your task:
Use the –query parameter of the Get-WMIObject cmdlet to retrieve *only* the build number and service pack major version number of the local operating system. Do not use the Select-Object cmdlet at all.

Here’s a tip:
Try the Win32_OperatingSystem class

Here’s a tip:
Remember that a WQL query can specify a list of properties to retrieve.

Lab 8-1
Using Pipeline Filtering and Operators

Task #1

Here's your task:
Use Get-Process to display a list of processes which are responding to Windows. Sort the list alphabetically by process name.

Here's a tip:
Pipe process to Get-Member to see their properties. Is there a property that tells you if the process is responding to Windows?

Task #2

Here's your task:
Display a list of services which are configured to start automatically, but which are not currently running.

Here's a tip:
You can't use Get-Service for this – you'll have to get a list of services using another technique.

Here's a tip:
The Win32_Service WMI class has a StartMode and Status property which will be useful.

Here's a tip:
Because you want to examine two properties, you'll need to use a complex expression along with Where-Object.

Task #3

Here's your task:
Working with the same command-line that you did for Task #2, extend the command-line to produce its results in an HTML file.

Here's a tip:
You may need to ask for Help with *HTML* to see what cmdlets are available for converting objects to HTML.

Here's a tip:
Converting to HTML doesn't imply creating a file; you'll need to pipe the HTML to another cmdlet that writes the HTML text to a file.

Task #4

Here's your task:
Use the WMI Win32_Process class to retrieve a list of processes. Use Where-Object to filter out all processes that aren't named "Notepad." For the processes that are remaining, execute their Terminate() method.

Here's a tip:
You can't use Get-Process for this – using the WMI class will allow this command-line to be repurposed to run against a remote computer.

Here's a tip:
Run Notepad before testing your command-line to ensure that it works.

Here's a tip:
Before executing the Terminate() method, try just writing the process' name. This will allow you to verify correct operation of your command-line since the Terminate() method can't be used with a "-whatIf" switch.

Task #5

Here's your task:
Create a text file named C:\Test\Computers.txt. Put two computer names into the file: Localhost, and your computer's name.

Write a command-line that will read the names from the file, connect to each computer using WMI, and retrieve each computer's Win32_LogicalDisk class.

Here's a tip:
Get-Content will retrieve the contents of a text file, and make each line of the file into an object.

Here's a tip:
You can't pipe a collection of names directly to Get-WMIObject. You'll need to work with each name one at a time.

Lab 10-1
Using More Key Cmdlets

Task #1

Here’s your task:
What are the default properties, and what is the default layout, used by Get-Service? Are all possible properties of the Service objects displayed?

Here’s a tip:
Try running the Get-Service cmdlet. Piping a service object to Get-Member will show you all of its properties.

Task #2

Here’s your task:
Create a table of running processes that lists the process name, process ID, and whether or not the process is responding to Windows. Sort the list so that responding processes are listed last, in alphabetical order.

Here’s a tip:
You’ll need to use three different cmdlets in your pipeline.

Task #3

Here’s your task:
Create a compact list of services which are running, displaying only the services’ names in a multi-column format.

Here’s a tip:
Remember, there are three main formatting cmdlets that you’ve been shown.

Here’s a tip:
You’ll need to use three cmdlets in this pipeline.

Task #4

Here's your task:
Create a list of all services on your computer. Display every possible property name for each service, and ensure that the output is legible.

Here's a tip:
You can use wildcards with the property list in a Format-* cmdlet.

Here's a tip:
There may be too many properties for a table-style layout.

Task #5

Here's your task:
Write a one-liner that asks you to "Type your name," and then displays your name in red text against a black background.

Here's a tip:
You can pipe things to Write-Host.

Here's a tip:
Whatever you enter into Read-Host is passed down the pipeline.

Here's a tip:
Ask for Help with Write-Host.

Task #6

Here's your task:
Run **5+5** from the command-line. Does running **Write-Output 5+5** produce the same result? What about doing this with **Write-Host**? If there's a difference, why do you think that is?

Task #7

Here's your task:
Taking what you learned from Task #6, take an expression like **5+5** and make PowerShell display it in yellow text.

Here's a tip:
Anything that is evaluated at the command-line is placed into the pipeline… it's only displayed because the pipeline "ends" in Out-Default. What could you put into the middle of the pipeline to force the text to be yellow?

Lab 12-1
Using Objects, Variables, Arrays, and Escapes

Task #1

Here’s your task:
Using the simplest command-line possible, display the properties for a String object.

Here’s a tip:
Get-Member can show you the properties for any object.

Task #2

Here’s your task:
Create a new variable named $var. Place “12345” into it, ensuring that “12345” is treated as a string. Pipe $var to Get-Member to verify that it is a string.

Here’s a tip:
You’ll need to explicitly type the variable $var.

Task #3

Here’s your task:
Create a new variable named $var and place “Hello World” into it. Have PowerShell tell you if the object contained in $var starts with the letter “X” or not.

Here’s a tip:
String types have a method called StartsWith() that you can execute.

Task #4

Here's your task:
Create an array named $arr, containing the words "one," "two," and "three." Display only the elements of $arr which are fewer than four characters in length, and have them displayed in green.

Here's a tip:
You'll need to use three cmdlets in your final command-line in order to make this work.

Here's a tip:
String types have a property which indicated how many characters the string contains. Pipe a string to Get-Member to try and locate this property and discover its name.

Task #5

Here's your task:
Create a variable named $var and have it contain the phrase "Hello World." Then, create a second variable named $var2 which contains the phrase "I say," a space, and then the contents of $var.

Here's a tip:
You didn't learn this yet, but when PowerShell sees a variable inside *double quotes*, it replaces the variable with its contents. It does not do this for *single quotes.*

Lab 13-1
Using Scope

Task #1

Do this first:
Create a script named ScopeTest.ps1. In it, place the following code:

```
$result = $var1 + $var1
Write "The result is $result"
```

Save the script.

Next, do this:
Open a new instance of PowerShell.

From within the shell, run your script. What result did you get? Since $var1 hasn't been defined in any scope, it contains no value by default.

Now, do this:
In the shell, run this command:

```
$var1 = 5
```

Run your script again. What result did you get? Since $var1 hasn't been defined in the script's scope, it uses the $var1 from the global scope.

Why this is important:
You've learned that the exact same script can return different results depending on the condition of the Global scope. If your script is run on a computer other than your own, you can't guarantee the contents of the Global scope – so it's a bad idea to rely upon it.

Now, do this:
Modify your ScopeTest.ps1 script and include the following as the first line of code:

```
$var1 = 2
```

Finally, do this:
Run your script again. What results did you get? Your script defines $var1 in its own scope, so it should get the same results every time you run it.

Try modifying the value of $var1 in the shell, and run your script again. Your script should return the same consistent results every time, no matter what is in the Global scope.

Lab 14-1
Using Advanced Operators

Task #1

Here's your task:
How could you check to see if the contents of a variable named $var contained the phrase "exec"?

Here's a tip:
There's a comparison operator that allows you to use wildcards.

Task #2

Here's your task:
Write a one-liner that will list all files and folders on your computer that contain the phrase "exec".

Here's a tip:
You'll need to use a cmdlet that can return all the files and folders on your computer and put them into the pipeline.

Here's a tip:
You can pipe files and folders to Get-Member to see what properties they have.

Here's a tip:
You'll need to use a second cmdlet that can remove pipeline items which don't meet your criteria.

Task #3

Here's your task:
Extend your solution from Task #2 to only show files that are less than 200 bytes.

Here's a tip:
This is where to use the logical operators you've learned about.

Task #4

Here's your task:
Create a list of all files on your computer that are larger than 100MB.

Here's a tip:
Repurpose your work from Task #2 and Task #3 to complete this task.

Lab 15-1
Using Scripting Constructs

Task #1

Here's your task:
Write a script that prompts the user for an event log filename, and then retrieves the newest 50 events from that event log.

For each event, examine the category. If the category is (0), output the event's "time written" in green. If the category is (101), output the time in yellow. If the category is (103), output in Red. For all other categories, output in white.

Here's a tip:
Use Get-Member to find the property names you need.

Use Get-EventLog, piped to Format-List, to display all the properties so that you can see what values go into these properties.

Use the scripting constructs you just learned, and take things step-by-step to construct a script that accomplishes the task as described.

Lab 16-1
Using Modularization Techniques

Task #1

Here's your task:
Write a simple function named Get-OSBuild that accepts a computer name, and outputs the computer's operating system build number. This build number is a property of the Win32_OperatingSystem WMI class.

SOLUTIONS

Lab 2-1 Solutions

Task #1

Here's the solution:
Run this command:

```
Dir "C:\Program Files\" -recurse
```

Task #2

Here's the solution:
Run this command:

```
Dir HKCU:\SOFTWARE -recurse
```

Task #3

Here's the solution:
Run this command:

```
Type ENV:\Systemroot
```

Here's an extra fact:
The ENV: drive is also exposed through a *namespace* named $env:. This is done to make accessing environment variables easier. Try running this:

```
$env:systemroot
```

Task #4

Here's the solution:
Run these commands:

```
CD HKCU:\Software
CP Intel SAPIENClass
Dir
```

This assumes that you have an existing key named Intel; if not, substitute another key. Ideally, use a smaller key, rather than a large key such as Microsoft. The **Dir** command allows you to verify that the new registry key was created.

To copy the key's contents, you'd add the **–recurse** parameter to the **CP** (**Copy-Item**) cmdlet.

Task #5

Here's the solution:
Run these commands:

```
CD HKCU:\Software
RD SAPIENClass
Dir
```

The **Dir** command allows you to verify that the new registry key was created.

Lab 3-1 Solutions

Task #1

Here's the solution:
Run this command:

```
Help gci
```

Task #2

Here's the solution:
Run this command:

```
New-Alias d Get-ChildItem
```

Task #3

Here's the solution:
We started by running **Get-PSSnapIn** to get a list of snap-ins. We guessed that Microsoft.PowerShell.Management might contain Get-WMIObject, so we ran **Get-Command –psSnapIn Microsoft.PowerShell.Management** and found that Get-WMIObject was indeed listed.

Task #4

Here's the solution:

```
Start-Transcript
Stop-Transcript
```

Task #5

Here's the solution:
Run one of these commands:

```
Stop-Process Calc
Kill –name Calc
```

Task #6

Here’s the solution:
Run this command:

```
Get-Content c:\computers.txt –totalCount 10
```

Task #7

Here’s the solution:
Run this command:

```
Get-Date –displayHint date
```

Task #8

Here’s the solution:
Run this command:

```
Get-EventLog Security –newest 20
```

Lab 4-1 Solutions

Task #1

Here’s the solution:
Run this command:

```
Get-Service | Get-Member
```

Task #2

Here’s the solution:
Run this command:

```
Get-Process | Out-File c:\processes.txt
```

Task #3

Here’s the solution:
Run this command:

```
Get-Service | Stop-Service -whatIf
```

Task #4

Here’s the solution:
Run this command:

```
Get-Service | Stop-Service -confirm
```

Task #5

Here’s the solution:
Run this command:

```
Get-Content C:\Files.txt | Copy-Item –dest C:\Test
```

Lab 5-1 Solutions

Task #1

Here's the solution:
Run this command:

```
Get-Service | Get-Member
```

Task #2

Here's the solution:
Run this command:

```
Get-EventLog System | Get-Member
```

Task #3

Here's the solution:
Run this command:

```
Get-Process | Sort-Object Handles –desc | Select-Object –first 10
```

Task #4

Here's the solution:
Run these commands:

```
Get-Service | Group-Object Status
```

Task #5

Here's the solution:
Run these commands:

```
Get-Service | Measure-Object
```

Task #6

Here’s the solution:
Run this command:

```
Get-Process | Measure-Object Handles –average –min -max
```

Task #7

Here’s the solution:
Run these commands:

```
Get-EventLog Security | Select EventID,TimeGenerated
```

Lab 6-1 Solutions

Task #1

Here's the solution:
Run this command:

```
Set-ExecutionPolicy RemoteSigned
```

Task #2

Here's the solution:
Run this command:

```
./test
```

Task #3

Here's the solution:

```
Get-AuthenticodeSignature
Set-AuthenticodeSignature
```

Task #4

Here's the solution:
You need to create a personal PowerShell profile.

In your personal Documents folder ("Documents" or "My Documents"), create a folder named WindowsPowerShell (all one word). In that folder, create a new text file named **Microsoft.PowerShell_profile.ps1**. Be careful – if Windows is configured to hide filename extensions, it may add a "hidden" .TXT filename extension to the file, and it won't work.

In this file, place the following code:

```
CD C:\TEST
```

Open the shell and verify that it is in the C:\Test folder when it starts.

Task #5

Here’s the solution:
You need to modify your PowerShell profile. Add the following code:

```
$cred = Get-Credential Administrator
```

Verify that you can restart the shell and it prompts you for a password. Then run:

```
$cred
```

To verify that the credential was created. The $cred variable can now be passed to any –credential parameter.

Lab 7-1 Solutions

Task #1

Here's the solution:
Win32_Service
Win32_LogicalDisk
Win32_NetworkAdapterConfiguration
Win32_Process
Win32_OperatingSystem

Task #2

Here's the solution:
Run this command:

```
Get-WMIObject Win32_LogicalDisk
```

Task #3

Here's the solution:
Run this command:

```
Get-WMIObject Win32_LogicalDisk | Sort-Object DriveType
```

Task #4

Here's the solution:
Run this command:

```
Get-WMIObject Win32_Service -filter "State = 'Running'"
```

Task #5

Here's the solution:
Run this command:

```
Get-WMIObject Win32_Service -filter "State = 'Running'" `
 | Select Name,StartMode
```

Task #6

Here's the solution:
Run this command:

```
Get-WMIObject –query "SELECT * FROM Win32_LogicalDisk WHERE `
 DriveType = 3"
```

Task #7

Here's the solution:
Run this command:

```
Get-WMIObject –query "SELECT BuildNumber,ServicePackMajorVersion `
 FROM Win32_OperatingSystem"
```

Lab 8-1 Solutions

Task #1

Here's the solution:
Run this command:

```
Get-Process | Where-Object { $_.Responding } | `
 Sort-Object Name
```

Task #2

Here's the solution:
Run this command:

```
Get-WMIObject Win32_Service | Where-Object `
 { $_.StartMode -eq "Auto" -and $_.State -ne "Running" }
```

Task #3

Here's the solution:
Run this command:

```
Get-WMIObject Win32_Service | Where-Object `
 { $_.StartMode -eq "Auto" -and $_.State -ne "Running" } | `
 ConvertTo-HTML | Out-File C:\Test\NotRunning.html
```

Task #4

Here's the solution:
Run this command:

```
Get-WMIObject Win32_Process | Where-Object `
 { $_.Name -eq "Notepad.exe" } | ForEach-Object { $_.Terminate() }
```

Task #5

Here’s the solution:
Run this command:

```
Get-Content C:\Computers.txt | `
 ForEach-Object { Get-WMIObject Win32_LogicalDisk `
 -computerName $_ }
```

Lab 10-1 Solutions

Task #1

Here's the solution:
Name, DisplayName, and Status are the default properties. The default layout is a table. These are not the only properties of a service:

```
Get-Service | Get-Member
```

Task #2

Here's the solution:
Run this command:

```
Get-Process | Sort-Object Responding,Name | `
 Format-Table Name,ID,Responding
```

Task #3

Here's the solution:
Run this command:

```
Get-Service | Where-Object {$_.Status –eq "Running"} | `
 Format-Wide Name
```

Task #4

Here's the solution:
Run this command:

```
Get-Service | Format-List *
```

Task #5

Here's the solution:
Run this command:

```
Read-Host "Enter your name" | Write-Host –fore red –back black
```

Task #6

Here's the solution:
Write-Host and Write-Output do not evaluate expressions – they simply output what you give them. When you just type an expression at the shell's prompt, the expression is evaluated and the result is placed into the success pipeline.

Task #7

Here's the solution:
Run this command:

```
5 + 5 | Write-Host –fore yellow
```

Lab 12-1 Solutions

Task #1

Here's the solution:
Name, DisplayName, and Status are the default properties. The default layout is a table. These are not the only properties of a service:

```
"x" | gm
```

Task #2

Here's the solution:
Run this command:

```
[string]$var = 12345
```

Task #3

Here's the solution:
Run this command:

```
$var = "Hello World"
$var.StartsWith("X")
```

Task #4

Here's the solution:
Run this command:

```
$arr = @("one","two","three")
$arr | where { $_.Length –lt 4 } | Write-Host –fore green
```

Task #5

Here's the solution:
Run this command:

```
$var = "Hello World"
$var2 = "I say, $var"
$var2
```

Note that this does not work the same:

```
$var2 = 'I say, $var'
$var2
```

Lab 14-1 Solutions

Task #1

Here's the solution:
Run this command:

```
$var –like "*exec*"
```

Task #2

Here's the solution:
Run this command:

```
Get-ChildItem –recurse | Where { $_.FullName –like "*exec*" }
```

Task #3

Here's the solution:
Run this command:

```
Get-ChildItem –recurse | Where { $_.FullName –like "*exec*" `
 -and $_.Length –lt 200 }
```

Task #4

Here's the solution:
Run this command:

```
Get-ChildItem –recurse | Where { $_.Length –gt 100MB }
```

Lab 15-1 Solutions

Task #1

Here’s the solution script:

```
$log = Read-Host "Which event log to get?"
$events = Get-EventLog $log -newest 50
foreach ($event in $events) {
      switch ($event.category) {
            "(0)"   { $color = "Green" }
            "(101)" { $color = "Yellow" }
            "(103)" { $color = "Red" }
            default { $color = "White" }
      }
      Write-Host $event.timewritten -fore $color
}
```

Lab 16-1 Solutions

Task #1

Here's the solution:
Create a script containing this:

```
function Get-OSBuild {
      param($computer)
      $wmi = gwmi win32_operatingsystem -ComputerName $computer
      foreach ($os in $wmi) {
            write $os.buildnumber
      }
}

Get-OSBuild localhost
```