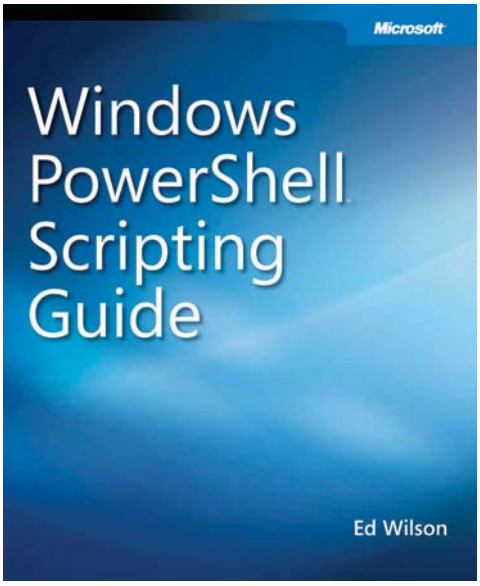

Windows PowerShell™ Scripting Guide

Ed Wilson

PREVIEW CONTENT This excerpt contains uncorrected manuscript from an upcoming Microsoft Press title, for early preview, and is subject to change prior to release. This excerpt is from *Windows PowerShell™ Scripting Guide* from Microsoft Press (ISBN 978-0-7356-2279-1, copyright 2008 Ed Wilson, all rights reserved), and is provided without any express, statutory, or implied warranties

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/9541.aspx

Microsoft Press

978-0-7356-2279-1

Table of Contents

Chapter 2
Scripting Windows PowerShell

After completing this chapter, you will be able to:

- Configure the scripting policy for Windows PowerShell
- Run Windows PowerShell scripts
- Use Windows PowerShell flow control statements
- Use decision-making and branching statements
- Identify and work with data types
- Use regular expressions to provide advanced matching capabilities
- Use command-line arguments

On the CD All the scripts used in this chapter are located on the CD that accompanies this book in the \scripts\chapter02 folder.

Why Use Scripting?

For many network administrators writing scripts—any kind of scripts—is a dark art more akin to reading tea leaves than administering a server. Indeed, while most large corporations seem to always have a "scripting guy," they rarely have more than one. This is in spite of the efforts by Microsoft to promote Visual Basic Scripting Edition (VBScript) as an administrative scripting language. While most professionals will agree that the ability to quickly craft a script to make ad hoc changes to dozens of networked servers is a valuable skill, few actually possess this skill. In reality, however, many of the corporate "scripting guy" skills are more akin to knowing where to find a script that can easily be modified than to actually understanding how to write a script from scratch.

Hopefully, this will change in the Windows PowerShell world. The Windows PowerShell syntax was deliberately chosen to facilitate ease of use and ease of learning. Corporate enterprise Windows administrators are the target audience.

So why use scripting? There are several reasons. First, a script makes it easy to document a particular sequence of commands. If you need to produce a listing of all the shares on a computer, you can use the *Win32_share* WMI class and the Get-WmiObject cmdlet to retrieve the results, as shown here:

```
PS C:\> Get-wmiObject win32_share


Name                    Path                    Description
----                    ----                    -----------
ADMIN$                  C:\Windows              Remote Admin
```

```
C$                      C:\                         Default share

CCMLogs$                C:\Windows\system32\ccm\logs

CCMSetup$               C:\Windows\system32\ccmsetup

IPC$                                                                Remote
IPC

music                   C:\music                    none

VPCache$                C:\Windows\system32\VPCache

WMILogs$                C:\Windows\system32\wbem\logs
```

But, suppose you only want to have a list of file shares? You may not be aware that a file share is a type 0 share. So perhaps you need to search for this information on the Internet. Once you have obtained the information, use the modified command shown here:

```
PS C:\> Get-WmiObject win32_share -Filter "type = '0'"


Name                    Path                        Description

----                    ---                         -----------

CCMLogs$                C:\Windows\system32\ccm\logs

CCMSetup$               C:\Windows\system32\ccmsetup

music                   C:\music                     none

VPCache$                C:\Windows\system32\VPCache

WMILogs$                C:\Windows\system32\wbem\logs
```

You can see that not only do you need to remember the share type of 0, but the syntax is a bit more complicated as well. So where do you write down this information? Here's one suggestion: When I was an administrator working on the Digital VAX, I kept a small pocket-size notebook to store such cryptic commands. Of course, if I ever lost my little notebook or failed to carry it, I was in big trouble!

Now suppose you are only interested in file shares that do not have a description assigned to them. This command is shown here:

```
PS C:\> Get-WmiObject win32_share -Filter "type = '0' AND description = ''"


Name                    Path                        Description

----                    ----                        -----------

CCMLogs$                C:\Windows\system32\ccm\logs

CCMSetup$               C:\Windows\system32\ccmsetup

VPCache$                C:\Windows\system32\VPCache

WMILogs$                C:\Windows\system32\wbem\logs
```

At this point, you may feel the command and associated syntax are complicated enough to justify writing a script. Creating the script is easy; simply copy it from the Windows PowerShell console and paste it into a text file. Name the script and change the extension to .ps1. You can then run the script from inside Windows PowerShell. The commands just

shown are saved in Share.txt in the chapter02 folder on the companion CD-ROM. The script is named GetFileShares.ps1.

An additional advantage to configuring a command as a script is that you can easily make modifications. Whereas the previous command was limited to reporting only on file shares, you can make a change to the script to allow reporting on print shares, remote administrative shares, IPC shares, or any other defined share type. You can modify the script so you can choose a share type when you launch the script. To do this, use an *if … else* statement to see if a command-line argument has been supplied to the script.

Tip To check for a command-line argument, look for *$args,* which is the automatic variable created to hold command-line arguments.

If there is a command-line argument, use the value supplied to the command line. If no value is supplied when the script is launched, then you must supply a default value to the script. For this script, you will list file shares and inform the user that you are using default values. The Get-WmiObject syntax is the same as you used previously in the VBScript days. When writing a script, it's also useful to display a *usage string*. The following script, GetSharesWithArgs.ps1, includes an example command to assist you with typing the correct syntax for the script.

GetSharesWithArgs.ps1

```
if($args)
 {
   $type = $args
   Get-WmiObject win32_share -Filter "type = $type"
 }
ELSE
 {
  Write-Host
  "
  Using defaults values, file shares type = 0.
  Other valid types are:
  2147483651 for disk drive admin share
  2147483649 for print queue admin share
  2147483650 for device admin share
  2147483651 for ipc$ admin share
  Example: C:\GetSharesWithArgs.ps1 '2147483651'
  "
  $type = '0'
  Get-WmiObject win32_share -Filter "type = $type"
 }
```

Another reason why network administrators write Windows PowerShell scripts is to run the script as a scheduled task. In the Windows world there are multiple task scheduler engines. Using the *Win32_ScheduledJob* WMI class you can create, monitor, and delete scheduled jobs. This WMI class has been available since the Windows NT 4.0 days. Both Windows XP and Windows Server 2003 have the Schtasks.exe utility, which offers more flexibility than the *Win32_ScheduledJob* WMI class. Besides Schtasks.exe, Windows Vista and Windows Server 2008 also include the *Schedule.Service* object to simplify the configuration of scheduled jobs.

The script, ListProcessesSortResults.ps1, is something you may want to schedule to run several times daily. The script produces a list of currently running processes and writes the results to a text file as a formatted and sorted table.

ListProcessesSortResults.ps1

```
$args = "localhost","loopback","127.0.0.1"


foreach ($i in $args)
   {$strFile = "c:\mytest\"+ $i +"Processes.txt"
    Write-Host "Testing" $i "please wait ...";
    Get-WmiObject -computername $i -class win32_process |
    Select-Object name, processID, Priority, ThreadCount, PageFaults,
      PageFileUsage |
    Where-Object {!$_.processID -eq 0} | Sort-Object -property name |
    Format-Table | Out-File $strFile}
```

Configuring the Scripting Policy

Since scripting in Windows PowerShell is not enabled by default, it is important to verify the level of scripting support provided on the platform before deployment of either scripts or commands. If you attempt to run a Windows PowerShell script when the support has not been enabled, you'll receive an error message and the script won't run. This error message is shown in Figure 2-1.

This is referred to as the *restricted execution policy*. There are four levels of execution policy that can be configured in Windows PowerShell with the Set-ExecutionPolicy cmdlet. These four levels are listed in Table 2-1. The restricted execution policy can be configured via Group Policy by using the Turn On Script Execution Group Policy setting in Active Directory directory service. It can be applied to either the computer object or to the user object. The computer object setting takes precedence over other settings.

> **Tip** To retrieve the script execution policy use the Get-ExecutionPolicy cmdlet.

Configure user preferences for the restricted execution policy with the Set-ExecutionPolicy cmdlet but note that these preferences won't override settings configured by Group Policy. Obtain the resulting set of restricted execution policy settings by using the Get-ExecutionPolicy cmdlet.

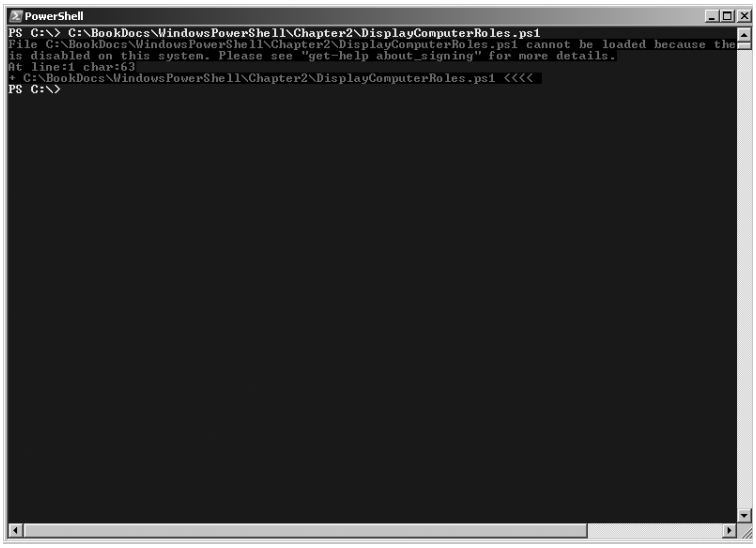

Figure 2-1 Attempting to run a script before scripting support is enabled generates an error.

Table 2-1 Script Execution Policy Levels

Level	Meaning
Restricted	Will not run scripts or configuration files.
AllSigned	All scripts and configuration files must be signed by a trusted publisher.
RemoteSigned	All scripts and configuration files downloaded from the Internet must be signed by a trusted publisher.
Unrestricted	All scripts and configuration files will run. Scripts downloaded from the Internet will prompt for permission prior to running.

You should be aware that on Windows Vista, access to the registry key that contains the script execution policy is restricted. A "normal" user will not be allowed to modify the key, and even an administrator running with User Account Control (UAC) turned on will not be allowed to modify the setting. If modification is attempted, the error shown in Figure 2-2 will be generated.

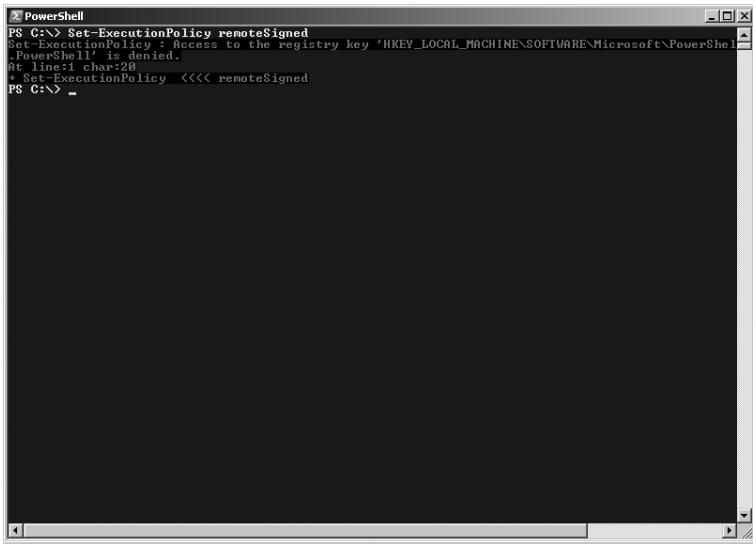

Figure 2-2 An attempt to run the Set-ExecutionPolicy cmdlet will fail if the user does not have administrative rights.

There are, of course several ways around the UAC issue. One choice is to simply turn off UAC; in most circumstances this is an undesirable solution. A better solution is to right-click the Windows PowerShell icon and select Run As Administrator as shown in Figure 2-3.

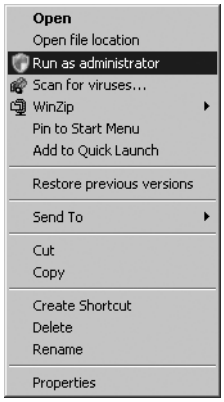

Figure 2-3 To launch Windows PowerShell with administrative rights, you can right-click the icon, and select Run As Administrator.

If you find right-clicking a bit too time-consuming (as I do!) you might prefer to create a second Windows PowerShell shortcut. You might name this second shortcut admin_ps and configure the shortcut properties to launch with administrative rights. For about 90 percent of all your administrative needs, the first shortcut should suffice. If, however, you need "more power," then choose the administrative one. The shortcut properties you can use for the admin_ps "administrative PowerShell" shortcut are shown in Figure 2-4.

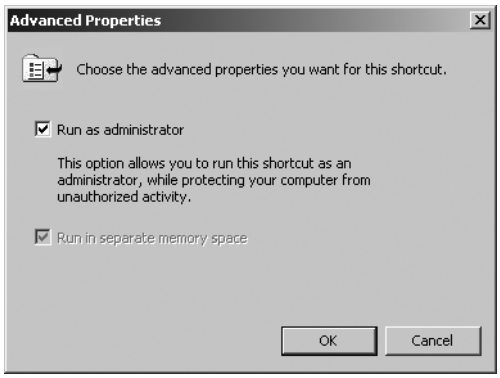

Figure 2-4 To configure the Windows PowerShell shortcut to run with administrative rights, choose the Run As Administrator check box found under Advanced Properties.

Running Windows PowerShell Scripts

You can't simply double-click a Windows PowerShell script and have it run. You cannot type the name in the Start | Run dialog box, either. If you are inside Windows PowerShell, you can run scripts if you have enabled the execution policy, but you need to type the entire path to the script you want to run and make sure to include the .ps1 extension.

If you need to run a script from outside Windows PowerShell, you must type the full path to the script, but you must also feed it as an argument to the PowerShell.exe program. In addition, you probably want to specify the *-noexit* switch so you can read the output from the script inside the Windows PowerShell console. This syntax is shown in Figure 2-5.

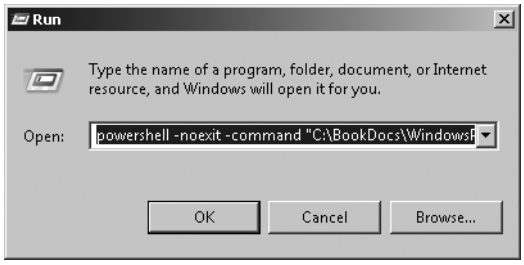

Figure 2-5 To run a Windows PowerShell script from outside the console, use the -noexit argument to allow you to see the results of the script.

Use of Variables

When working with Windows PowerShell, the default is that you don't need to declare variables prior to use; the variable is declared when you use it to hold data. All variable names must be preceded with a dollar sign. There are a number of special variables in Windows PowerShell. These variables are created automatically and each has a special meaning. Table 2-2 lists the special variables and their associated meanings.

Table 2-2 Use of Special Variables

Name	Use
*$^*	Contains the first token of the last line input into the shell.
$$	Contains the last token of the last line input into the shell.
$_	The current pipeline object; used in script blocks, filters, Where-Object, ForEach-Object, and switch.
$?	Contains the success/fail status of the last statement.
$args	Used in creating functions requiring parameters.
$error	If an error occurred, the *error* object is saved in the *$error* variable.
$executioncontext	The *execution* objects available to cmdlets.
$foreach	Refers to the enumerator in a *foreach* loop.
$home	The user's home directory; set to %HOMEDRIVE%\%HOMEPATH%.
$input	Input is piped to a function or code block.
$match	A hash table consisting of items found by the *-match* operator.
$myinvocation	Information about the currently executing script or command line.
$pshome	The directory where Windows PowerShell is installed.
$host	Information about the currently executing host.
$lastexitcode	The exit code of the last native application to run.
$true	Boolean TRUE.
$false	Boolean FALSE.
$null	A null object.
$this	In the types.ps1xml file and some script block instances this represents the current object.

$ofs	Output field separator used when converting an array to a string.
$shellid	The identifier for the shell. This value is used by the shell to determine the execution policy and what profiles are run at startup.
$stacktrace	Contains detailed stack trace information about the last error.

Use of Constants

Constants in Windows PowerShell are like variables with two important exceptions: Their value never changes, and they cannot be deleted. Constants are created by using the Set-Variable cmdlet and specifying the *-option* argument to be equal to constant.

BestPractices When referring to a constant in the body of the script, you must preface it with the dollar sign—just like any other variable. However, when creating the constant (or even a variable) by using the Set-Variable cmdlet, as you specify the *name* argument you don't include a dollar sign.

In the GetHardDiskDetails.ps1 script that follows, there is a constant named *$intDriveType* with a value of 3 assigned. This constant is used because the *Win32_LogicalDisk* WMI class uses a value of 3 in the *disktype* property to describe a local fixed disk. When using Where-Object and a value of 3, you eliminate network drives, removable drives, and ram drives from the items returned.

The *$intDriveType* constant is only used with the Where filter line. The value of *$strComputer*, however, will change once for each computer name that is specified in the array *$aryComputers*. In the GetHardDiskDetails.ps1 script, the value of *$strComputer* will change twice. The first time through the loop it will be equal to "loopback" and the second time through the loop it will be equal to "localhost." Even if you add 250 different computer names, the effect will be the same—the value of *$strComputer* will change each time through the loop.

GetHardDiskDetails.ps1

```
$aryComputers = "loopback", "localhost"

Set-Variable -name intDriveType -value 3 -option constant


foreach ($strComputer in $aryComputers)


   {"Hard drives on: " + $strComputer

   Get-WmiObject -class win32_logicaldisk -computername $strComputer|

      Where {$_.drivetype -eq $intDriveType}}
```

Using Flow Control Statements

Once scripting support is enabled on Windows PowerShell, you have access to some advanced flow control cmdlets. However, this does not mean you cannot do flow control inside the console. You can certainly use flow control statements inside the console. This is shown here:

```
PS C:\> Get-Process | foreach ( $_.name ) { if ( $_.name -eq "system" ) {
Write-Host "system process is ID : " $_.ID } }
```

The problem is the amount of typing. It may be preferable to save such a command in a script. Besides saving a long command in a file, there is also an advantage in readability. For example, you can line up the curly brackets and the other components of the commands. You can also avoid hard-coding process names into the script and instead, save them as variables. This makes it easy to modify the script or even to write the script to accept command-line arguments. In the GetProcessById.ps1 script shown here, you can see these options exhibited.

GetProcessById.ps1

```
$strProcess = "system"
Get-Process |
foreach ( $_.name ) {
  if ( $_.name -eq $strProcess )
   {
    Write-Host "system process is ID : " $_.ID
   }
}
```

Adding Parameters to ForEach-Object

In the GetWmiAndQuery.ps1 script, the ForEach-Object cmdlet produces a listing from all the WMI classes that have names containing *usb*. This particular script is very useful in that it produces a listing of both the process name and associated process ID (PID). In addition, the GetProcessByID.ps1 script is a good candidate to modify to accept a command-line argument. Begin with the *list* switch from the Get-WmiObject cmdlet; you'll end up with a complete listing of all WMI classes in the default WMI namespace. Pipeline the resulting object into the Where-Object cmdlet and filter the result set by the *name* property when it is like the value contained in the variable *$strClass*.

Using the *Begin* Parameter

Use the *-begin* parameter of the ForEach-Object cmdlet to write the name used to generate the WMI class listings. This action does not affect the current pipeline object. In fact, neither the *-begin* parameter or the *-end* parameter interact with the current pipeline object. But they are great places to perform pre-processing and post-processing. The *-process* parameter is used to contain the script block that will interact with the

current pipeline object. This is the default parameter, and doesn't need to be named. The Get-WmiAndQuery.ps1 script is shown here.

GetWmiAndQuery.ps1

```
$strClass = "usb"
Get-WmiObject -List |
Where { $_.name -like "*$strClass*" } |
ForEach-Object -begin `
  {
    Write-Host "$strClass wmi listings"
        Start-Sleep 3
   } `
-Process `
   {
     Get-wmiObject $_.name
   }
```

In the ProcessUsbHub.ps1 script, the Get-WmiObject cmdlet retrieves instances of the *Win32_USBHub* class. Once we have a collection of *usb hub* objects, we pipeline the object to the ForEach-Object cmdlet. Suggestion: To make the script easier to read, line up all the *-begin*, *-process* and *-end* parameters on the left side of the script. However, you will have to use the "backtick" or grave accent to indicate line continuation.

Tip The environment variable %computername% is always available and can be used to extract the computer name for a script. An easy way to retrieve the value of this variable is to use the Get-Item cmdlet to grab the value from the env:\ psdrive. The *value* property contains the computer name. This is illustrated here: (Get-Item env:\computerName) value.

The *-begin* section uses a code block to write the name of computer using the Write-Host cmdlet. Use a sub-expression to get the computer name from the env:\ psdrive; use the %computername% variable and extract its value.

Using the *Process* Parameter

In the *-process* section, simply use the current pipeline object (indicated by the *$_* automatic variable) to print the *pnpDeviceID* property from the *Win32_USBHub* WMI class. Again, use the grave accent to indicate line continuation.

Using the *End* Parameter

The last section of the ProcessUsbHub.ps1 script contains the *-end* parameter. Use the Write-Host cmdlet to print a string that indicates the command completed, and use a sub-expression to print the value returned by the Get-Date cmdlet. The ProcessUsbHub.ps1 script is listed here.

ProcessUsbHub.ps1

```
Get-WmiObject win32_usbhub |

foreach-object `

-begin { Write-Host "Usb Hubs on:" $(Get-Item env:\computerName).value } `

-process { $_.pnpDeviceID}  `

-end { Write-Host "The command completed at $(get-date)" }
```

Using the *For* Statement

Similar to the ForEach-Object cmdlet, the *for* statement is used to control execution of a script block as long as a condition is true. Most of the time, you will use the *for* statement to perform an action a certain number of times. In the line of code that follows, notice the basic *for* construction. Use parentheses to separate the expression being evaluated from the code block contained in curly brackets. The evaluated expression is composed of three sections. The first section is a variable *$a*; you assign the value of 1 to it. The second section contains the condition to be evaluated. In the code shown here, as long as the variable *$a* is less than or equal to the number 3, the command in the code block section continues to run. The last section of the evaluation expression adds the number 1 to the variable *$a*. The code block is a simple printout of the word *hello*.

```
for ($a = 1; $a -le 3 ; $a++) {"hello"}
```

The PingArange.ps1 script shown here is a very useful little script because it can be used to ping a range of Internet protocol(IP) addresses and will tell you whether or not the computer is responding to Internet control messaging packets(ICMP) packets. This is helpful in doing network discovery or in ensuring a computer is talking to the network. The *$intPing* variable is set to 10 and defined as an integer. Next, the *$intNetwork* variable is assigned the string 127.0.0. and is defined as a string.

The *for* statement is used to execute the remaining code the number of times specified in the *$intPing* variable. The counter variable is created on the *for* statement line. This counter variable, named *$i*, is assigned the value of 1. As long as *$i* is less than or equal to the value set in the *$intPing* variable, the script will continue to execute. The final step, completed inside the evaluator section of the *for* statement, is to add one to the value of *$i*.

The code block begins with the curly bracket. Inside the code block, first create a variable named *$strQuery*; this is the string that holds the WMI query. Placing this in a separate variable makes it easier to use *$intNetwork* along with the *$i* counter variable; these are used to create a valid IP address for the WMI query that results in a ping.

The *$wmi* variable is used to hold the collection of objects that is returned by the Get-WmiObject cmdlet. By using the *optional query* argument of the Get-WmiObject cmdlet, you are able to supply a WMI query. The *statuscode* property contains the result of the ping operation. A 0 indicates success, any other number means the ping failed. To present this information in a clear fashion, use an *if … else* statement to evaluate the *statuscode* property.

PingArange.ps1

```
[int]$intPing = 10

[string]$intNetwork = "127.0.0."


for ($i=1;$i -le $intPing; $i++)

{

$strQuery = "select * from win32_pingstatus where address = '" +

$intNetwork + $i + "'"

   $wmi = get-wmiobject -query $strQuery

   "Pinging $intNetwork$i ... "

   if ($wmi.statuscode -eq 0)

      {"success"}

      else

         {"error: " + $wmi.statuscode + " occurred"}
```

Using Decision-Making Statements

The ability to make decisions to control branching in a script is a fundamental technique. In fact, this is the basis of automation. A condition is detected and evaluated, and a course of action is determined. If you are able to encapsulate your logic into a script, you are well on your way to having servers that monitor themselves. As an example, when you open Task Manager on the server, what is the first thing you do? I often sort the list of processes by memory consumption. The GetTopMemory.ps1 script, shown here, does this.

GetTopMemory.ps1

```
Get-Process |

Sort-Object workingset -Descending |

Select-Object -First 5
```

The GetTopMemory.ps1 script might be useful because it saves time in sorting a list. But what do you do next? Do you kill the top memory consuming process? If you do, then there is no decision to make. However, suppose you want to kill off only user mode processes that consume more than 100 MB of memory? That may be a more constructive and better choice. This will require some decision-making capability. Let us first examine the classic *if … elseif … else* decision structure.

Using *If … Elseif … Else*

The most basic decision-making statement is the *if … elseif … else* structure. This structure is easy to use because it is perfectly natural and is implied in normal conversation. For example, consider the following conversation between two American tourists in Copenhagen:

```
If ( sunny and warm )
  { go to NyHavn }
Elseif ( cloudy and cool )
  { go to Tivoli }
Else
  { take s-tog to Malmo }
```

Even if you don't speak Danish, you will be able to follow the conversation. If it is sunny and warm, then the tourists will go to NyHavn. The first condition evaluation is whether the weather is going to be sunny and warm. The condition is always enclosed in smooth parentheses. The script block that will be executed if the condition is true is in curly brackets. In this example, if the weather is sunny and warm, the tourists will go to NyHavn (a beautiful port with lots of outdoor cafes). However, if the weather is cloudy and cool, they will go to Tivoli (an amusement park in the center of Copenhagen). If neither of these conditions is true, for example, if it is raining or snowing, the tourists will take the train to Malmo (a city in Sweden famous for its shopping).

To use the GetServiceStatus.ps1 script, you will first obtain a listing of all the services on the computer. Do this by using the Get-Service cmdlet. Once you have a listing of the services, use the Sort-Object cmdlet to sort the list of services based on their status. Next, use *foreach* to walk through the collection of services. As you iterate through the services, use *if … elseif … else* to evaluate the status. If the service is stopped, use the color red to display the name and status. If the service is running, use green to display the name and status. If the service is in a different state (such as pause), default to yellow to display the name and status. A decision matrix such as this is very uopseful in allowing you to quickly scan a long list of services. The GetServiceStatus.ps1 script is shown here. The constant color values that can be used with the Write-Host cmdlet are detailed in the table that follows.

GetServiceStatus.ps1

```
Get-Service |
Sort-Object status -descending |
foreach {
  if ( $_.status -eq "stopped")
   {Write-Host $_.name $_.status -ForegroundColor red}
  elseif ( $_.status -eq "running" )
   {Write-Host $_.name $_.status -ForegroundColor green}
  else
```

```
   {Write-Host $_.name $_.status -ForegroundColor yellow}
}
```

Black	DarkBlue	DarkGreen	DarkCyan
DarkRed	DarkMagenta	DarkYellow	Gray
DarkGray	Blue	Green	Cyan
Red	Magenta	Yellow	White

Using Switch

In other programming languages, *switch* would be called the *select case statement*. The switch statement is used to evaluate a condition against a series of potential matches. In this way, it is essentially a streamlined *if … elseif* statement. When using the switch statement, the condition to be evaluated is contained in side parentheses. Then, each condition to be evaluated is placed inside a curly bracket within the code block. This is shown in the following command:

```
$a=5;switch ($a) { 4{"four detected"} 5{"five detected"} }
```

In the DisplayComputerRoles.ps1 script that follows, the script begins by using the *$wmi* variable to hold the object that is returned by using the Get-WmiObject cmdlet. The *domainrole* property of the *Win32_computersystem* class is returned as a coded value. To produce an output that is more readable, the switch statement is used to match the value of the *domainrole* property to the appropriate text value.

DisplayComputerRoles.ps1

```
$wmi = get-wmiobject win32_computersystem
"computer " + $wmi.name + " is: "
switch ($wmi.domainrole)
   {
   0 {"`t Stand alone workstation"}
   1 {"`t Member workstation"}
   2 {"`t Stand alone server"}
   3 {"`t Member server"}
   4 {"`t Back up domain controller"}
   5 {"`t Primary domain controller"}
   default {"`t The role can not be determined"}
   }
```

Evaluating Command-Line Arguments

Switch is ideally suited to evaluate command-line arguments. In the GetDriveArgs.ps1 script example that follows, you can use a function named *funArg* to evaluate the value of the automatic variable *$args*. This automatic variable contains arguments supplied to the command line when a script is run. This is a convenient variable to use when working with

command-line arguments. Switch is used to evaluate the value of *$args.* Four parameter arguments are allowed with this script. The *all* argument does a WMI query to retrieve basic information on all logical disks on the computer. The argument *c* is used to return only information about the C drive. An interesting trick: The floppy drive is typically enumerated first, and the second element in the array is the C drive. If this is not the case on your system, you can change it. The purpose of the script is simply to point out the use of switch to parse command-line arguments. Using the array element number is a nice way to retrieve WMI information in Windows PowerShell. The *free* argument is used to only return free disk space on the C drive.

The *help* argument is used to print a help statement. It uses a here string to make it easy to type in the help message. The help message displays the purpose of the script and several examples of command lines.

GetDriveArgs.ps1

```
Function funArg()

{

switch ($args)

{

 "all" { gwmi win32_logicalDisk }

 "c"   { (gwmi win32_logicaldisk)[1] }

 "free" { (gwmi win32_logicaldisk)[1].freespace }

 "help" { $help = @"

This script will print out the drive information for

All drives, only the c drive, or the free space on c:

It also will print out a help topic

EXAMPLE:

 >GetDriveArgs.ps1 all

   Prints out information on all drives

 >GetDriveArgs.ps1 c

   Prints out information on only the c drive

 >GetDriveArgs.ps1 free

   Prints out freespace on the c drive

"@ ; Write-Host $help }

}

}


#$args = "help"

funArg($args)
```

Using Switch Wildcards

One of the more interesting uses of the switch command is the use of wildcards. This can open up new opportunities to write clear and compact code that is both powerful and easy to implement. The SwitchIPconfig.ps1 script holds the results of the *ipconfig /all* command in the *$a* variable. Use switch with the *-wildcard* argument and feed it the text to parse inside the smooth parenthesis. Then, open the script block with the curly brackets and type the pattern to match. In this case, it is a simple *DHCP Server* phrase. In the script block that will execute when the pattern match is found, use the Write-Host cmdlet to print the current line inside the *switch* block. The interesting point is the use of the *$switch* automatic variable as the enumerator. Specify the current property and retrieve the current line that is processing. In this way, you can print the line you are interested in examining. The SwitchIPconfig.ps1 script is shown here.

SwitchIPConfig.ps1

```
$a = ipconfig /all


switch -wildCard ($a)
 {
   "*DHCP Server*" { Write-Host $switch.current }
 }
```

Using Switch with Regular Expressions

Unlike a normal *select case* statement, the *switch* statement has the ability to work with regular expressions. When looking for valuable information, you can use the switch statement to open a text file, read the file into memory, and then use regular expressions to parse the file. Regular expressions can be as simple as matching a particular word or phrase or as complicated as validating a legitimate e-mail address. The SwitchRegEx.ps1 script that follows examines a sample text file for two words: *test* and *good*. If either word is found, the entire line containing the matched word prints.

Following the *switch* statement, you can use the *-regex* parameter to indicate that you want to use regular expressions as the matching tool. The value to switch on, inside the smooth parentheses, is actually a sub-expression that opens and reads the text file. The $ in front of the curly brackets surrounding the path to a text file is the command to open and read the text file into memory. Open the switch with the curly brackets and place each pattern to match inside single quotations. The code block that will execute if the regular expression is matched is also contained in curly brackets, and in this example it is a simple write-host. Once again, use the *$switch* enumerator to retrieve the current line where the pattern match occurs.

SwitchRegEx.ps1

```
switch -regex (${c:\testa.txt})
{
 'test' {Write-Host $switch.current}
 'good' {Write-Host $switch.current}
}
```

The text of the TestA.txt file is shown here. This example will assist you in evaluating the output from the script.

Testa.txt

```
This was a test file.
This was a good file.
This was a good test file.
```

Perhaps a more useful example of using the regular expression feature of the switch statement is the VersionOfVista.ps1 script. Assign the string version to the *$strPattern* variable, and hold the output of the net config workstation command into the *$text* variable. Then, use the *-regex* parameter on the switch statement and feed it the content stored in the *$text* variable, and look for the pattern that is stored in the *$strPattern* variable. Once you find it, print the entire line by using the current property of the automatic variable *$switch*. The nice thing about this script is that it tells you what version of Windows Vista you have. The entire output from net config workstation command is 19 lines long. To compare results, here is a sample output from VersionOfVista.ps1.

```
Software version                    Windows Vista (TM) Enterprise
```

VersionOfVista.ps1

```
$strPattern = "version"
$text = net config workstation

switch -regex ($text)
{
  $strPattern { Write-Host $switch.current }
}
```

Working with Data Types

Windows PowerShell is a strongly typed language that acts as if it were typeless. This is because Windows PowerShell does a good job of detecting data types and acting on them accordingly. If something appears to be a string, Windows PowerShell will treat it as a string. As an example, consider these three statements:

```
PS C:\> 1 + 1

2

PS C:\> 12:00 + :30

Unexpected token ':00' in expression or statement.

At line:1 char:6

+ 12:00  <<<< + :30

PS C:\> a + b

The term 'a' is not recognized as a cmdlet, function, operable program,

or script file. Verify the term and try again At line:1 char:2 + a  <<<< + b

PS C:\>
```

Notice that only one statement completed without error—the one containing added 1 + 1. Windows PowerShell properly detected these as numbers and allowed the addition to proceed. However, it is impossible to add letters or time.

However, if you put the letters *a* and *b* within double quotation marks and then add them, you will notice that the action succeeds. This is shown here:

```
PS C:\> "a" + "b"

Ab
```

This behavior is not surprising, and in fact, is to be expected. Double quotation marks turn the letters *a* and *b* into string values and concatenates the two letters. You can see this if you pipeline the letter *a* into the Get-Member cmdlet as shown here. Notice that the first line of output indicates the letter *a* is an object of the type *system.string*. Also observe that there are many properties and methods you can use on a *system.string* object.

```
PS C:\> "a" | get-member



   TypeName: System.String


Name             MemberType            Definition

----             ----------            ----------

Clone            Method                System.Object Clone()

System.Int32 CompareTo(String strB)

Contains         Method                System.Boolean Contains(String value)

CopyTo           Method                System.Void CopyTo(Int32 sourceIndex, Char[]
```

```
destination, Int32 destinationIn

EndsWith         Method               System.Boolean EndsWith(String value),

System.Boolean EndsWith(String value,

Equals           Method               System.Boolean Equals(Object obj),

System.Boolean Equals(String value), Syste...

GetEnumerator    Method               System.CharEnumerator GetEnumerator()

GetHashCode      Method               System.Int32 GetHashCode()

GetType          Method               System.Type GetType()

GetTypeCode      Method               System.TypeCode GetTypeCode()

get_Chars        Method               System.Char get_Chars(Int32 index)

get_Length       Method               System.Int32 get_Length()

IndexOf          Method               System.Int32 IndexOf(Char value, Int32

startIndex, Int32 count), System.Int32...

IndexOfAny       Method               System.Int32 IndexOfAny(Char[] anyOf, Int32

startIndex, Int32 count), System....

Insert           Method               System.String Insert(Int32 startIndex, String

value)

IsNormalized     Method               System.Boolean IsNormalized(), System.Boolean

IsNormalized(NormalizationForm

LastIndexOf      Method               System.Int32 LastIndexOf(Char value, Int32

startIndex, Int32 count), System.I...

LastIndexOfAny   Method               System.Int32 LastIndexOfAny(Char[] anyOf, Int32
startIndex, Int32 count), Sys...

Normalize        Method               System.String Normalize(), System.String

Normalize(NormalizationForm normaliz...

PadLeft          Method               System.String PadLeft(Int32 totalWidth),

System.String PadLeft(Int32 totalWid...

PadRight         Method               System.String PadRight(Int32 totalWidth),

System.String PadRight(Int32 totalW...

Remove           Method               System.String Remove(Int32 startIndex, Int32

count), System.String Remove(Int...

Replace          Method               System.String Replace(Char oldChar, Char

newChar), System.String Replace(Stri...

Split            Method               System.String[] Split(Params Char[]

separator), System.String[] Split(Char[] ...

StartsWith       Method               System.Boolean StartsWith(String value),

System.Boolean StartsWith(String val...

Substring        Method               System.String Substring(Int32 startIndex),
```

```
System.String Substring(Int32 star...
ToCharArray      Method               System.Char[] ToCharArray(), System.Char[]
ToCharArray(Int32 startIndex, Int3...
ToLower          Method               System.String ToLower(), System.String
ToLower(CultureInfo culture)
ToLowerInvariant Method               System.String ToLowerInvariant()
ToString         Method               System.String ToString(), System.String
ToString(IFormatProvider provider)
ToUpper          Method               System.String ToUpper(), System.String
ToUpper(CultureInfo culture)
ToUpperInvariant Method               System.String ToUpperInvariant()
Trim             Method               System.String Trim(Params Char[] trimChars),
System.String Trim()
TrimEnd          Method               System.String TrimEnd(Params Char[]
trimChars)
TrimStart        Method               System.String TrimStart(Params Char[]
trimChars)
Chars            ParameterizedProperty System.Char Chars(Int32 index) {get
```

If you pipeline the number 1 into the Get-Member cmdlet, you will see that it is a *system.int32* object, with a smaller listing of methods available than is available with the string class:

```
PS C:\> 1 | get-member


   TypeName: System.Int32


Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     System.Int32 CompareTo(Int32 value), System.Int32
CompareTo(Object value)
Equals      Method     System.Boolean Equals(Object obj), System.Boolean
Equals(Int32 obj)
GetHashCode Method     System.Int32 GetHashCode()
GetType     Method     System.Type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode()
ToString    Method     System.String ToString(), System.String
ToString(IFormatProvider provider), System.String ToS...
```

Once you have figured out how to use Get-Member to verify the reason for the behavior of an object, you can use the *type constraint* objects to confirm an object of a specific data type. If you want 12:00 to be interpreted as a *date time* object, use the [datetime] type constraint to cast the string 12:00 into a *date time* object. This is shown here:

```
PS C:\> [datetime]"12:00" | get-member


   TypeName: System.DateTime


Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         System.DateTime Add(TimeSpan value)
AddDays              Method         System.DateTime AddDays(Double value)
AddHours             Method         System.DateTime AddHours(Double value)
AddMilliseconds      Method         System.DateTime AddMilliseconds(Double value)
AddMinutes           Method         System.DateTime AddMinutes(Double value)
AddMonths            Method         System.DateTime AddMonths(Int32 months)
AddSeconds           Method         System.DateTime AddSeconds(Double value)
AddTicks             Method         System.DateTime AddTicks(Int64 value)
AddYears             Method         System.DateTime AddYears(Int32 value)
CompareTo            Method         System.Int32 CompareTo(Object value),
System.Int32 CompareTo(DateTime value)
Equals               Method         System.Boolean Equals(Object value),
System.Boolean Equals(DateTime value)
GetDateTimeFormats   Method         System.String[] GetDateTimeFormats(),
System.String[] GetDateTimeFormats(IFormat...
GetHashCode          Method         System.Int32 GetHashCode()
GetType              Method         System.Type GetType()
GetTypeCode          Method         System.TypeCode GetTypeCode()
get_Date             Method         System.DateTime get_Date()
get_Day              Method         System.Int32 get_Day()
get_DayOfWeek        Method         System.DayOfWeek get_DayOfWeek()
get_DayOfYear        Method         System.Int32 get_DayOfYear()
get_Hour             Method         System.Int32 get_Hour()
get_Kind             Method         System.DateTimeKind get_Kind()
get_Millisecond      Method         System.Int32 get_Millisecond()
get_Minute           Method         System.Int32 get_Minute()
get_Month            Method         System.Int32 get_Month()
```

```
get_Second           Method         System.Int32 get_Second()
get_Ticks            Method         System.Int64 get_Ticks()
get_TimeOfDay        Method         System.TimeSpan get_TimeOfDay()
get_Year             Method         System.Int32 get_Year()
IsDaylightSavingTime Method         System.Boolean IsDaylightSavingTime()
Subtract             Method         System.TimeSpan Subtract(DateTime value),
System.DateTime Subtract(TimeSpan value)
ToBinary             Method         System.Int64 ToBinary()
ToFileTime           Method         System.Int64 ToFileTime()
ToFileTimeUtc        Method         System.Int64 ToFileTimeUtc()
ToLocalTime          Method         System.DateTime ToLocalTime()
ToLongDateString     Method         System.String ToLongDateString()
ToLongTimeString     Method         System.String ToLongTimeString()
ToOADate             Method         System.Double ToOADate()
ToShortDateString    Method         System.String ToShortDateString()
ToShortTimeString    Method         System.String ToShortTimeString()
ToString             Method         System.String ToString(), System.String
ToString(String format), System.String T...
ToUniversalTime      Method         System.DateTime ToUniversalTime()
Date                 Property       System.DateTime Date {get;}
Day                  Property       System.Int32 Day {get;}
DayOfWeek            Property       System.DayOfWeek DayOfWeek {get;}
DayOfYear            Property       System.Int32 DayOfYear {get;}
Hour                 Property       System.Int32 Hour {get;}
Kind                 Property       System.DateTimeKind Kind {get;}
Millisecond          Property       System.Int32 Millisecond {get;}Property
 System.Int32 Minute {get;}
Month                Property       System.Int32 Month {get;}
Second               Property       System.Int32 Second {get;}
Ticks                Property       System.Int64 Ticks {get;}
TimeOfDay            Property       System.TimeSpan TimeOfDay {get;}
Year                 Property       System.Int32 Year {get;}
DateTime             ScriptProperty System.Object DateTime {get=if
($this.DisplayHint -ieq  "Date")...
```

There is no reason to use Get-Member to determine the data type of a particular object if you are only interested in the name of the object. To do this, you can use the *getType()* method as shown here. In the first case, you confirm that 12:00 is indeed a string. In the

second case, you cast the string into a *datetime* data type, and confirm it by once again using the *getType()* method as shown here:

```
PS C:\> "12:00".getType()


IsPublic IsSerial Name                                     BaseType

-------- -------- ----                                     --------

True     True     String                                   System.Object



PS C:\> ([dateTime]"12:00").getType()


IsPublic IsSerial Name                                     BaseType

-------- -------- ----                                     --------

True     True     DateTime                                 System.ValueType
```

All of these commands are in the DataTypes.txt file found in the chapter02 folder on the companion CD-ROM. Additional data type aliases are shown in Table 2-3.

Table 2-3 Data Type Aliases

Alias	Type
[int]	32-bit signed integer
[long]	64-bit signed integer
[string]	Fixed length string of Unicode characters
[char]	A Unicode 16-bit character
[bool]	True False value
[byte]	An 8-bit unsigned integer
[double]	Double-precision 64-bit floating point number
[datetime]	DateTime data type
[decimal]	A 128-bit decimal value
[single]	Single precision 32-bit floating point number
[array]	An array of values
[xml]	XML objects
[hashtable]	A hashtable object (similar to a dictionary object)

Unleashing the Power of Regular Expressions

One of the interesting features of Windows PowerShell is the ability to work with regular expressions. Regular expressions are optimized to manipulate text. You've learned about using regular expressions with the *switch* statement to match a particular word, however, you can do as much with the *-wildcard* switch. Now you'll learn some of the more advanced tasks you can complete with regular expressions. Table 2-4 lists the escape sequences you can use with regular expressions.

Table 2-4 Escape Sequences

Character	Description
ordinary characters	Characters other than . $ ^ { [(|) * + ? \ match themselves.
\a	Matches a bell (alarm) \u0007.
\b	Matches a backspace \u0008 if in a [] character class; in regular expression is word boundary.
\t	Matches a tab \u0009.
\r	Matches a carriage return \u000D.
\v	Matches a vertical tab \u000B.
\f	Matches a form feed \u000C.
\n	Matches a new line \u000A.
\e	Matches an escape \u001B.
\040	Matches an ASCII character as octal (up to three digits); numbers with no leading zero are backreferences if they have only one digit or if they correspond to a capturing group number. For example, the character \040 represents a space.
\x20	Matches an ASCII character using hexadecimal representation (exactly two digits).
\cC	Matches an ASCII control character; for example, \cC is control-C.
\u0020	Matches a Unicode character using hexadecimal representation (exactly four digits).

The RegExTab.ps1 script illustrates using an escape sequence in a regular expression script. It opens a text file and looks for tabs. The easiest way to work with regular expressions is to store the pattern in its own variable. This makes it easy to modify and to experiment without worrying about breaking the script (simply use the # sign to comment out the line, then create a new line with the same name and a different value).

The RegExTab.ps1 script specifies \t as the pattern. According to Table 2-4 this means you look for tabs. Feed the pattern, contained in *$strPattern,* to the [regex] type accelerator as shown here:

```
$regex = [regex]$strPattern
```

Next, store the content of the TabLine.txt text file into the *$text* variable by using the syntax shown here:

```
$text = ${C:\Chapter02\tabline.txt}
```

Then, use the *matches* method to parse the text file and look for matches with the pattern specified in the *$strPattern*. Notice that you have already associated the pattern with the *regular expression* object in the *$regex* variable. Count the number of times you have a match. The complete Regextab.ps1 script is shown here.

RegExTab.ps1

```
$strPattern = "\t"

$regex = [regex]$strPattern


$text = ${C:\Chapter02\tabline.txt}


$mc = $regex.matches($text)

$mc.count
```

Table 2-5 lists the character patterns that can be used with regular expressions for performing advanced pattern matching.

Table 2-5 Character Patterns

Character	Description
[character_group]	Matches any character in the specified character group. For example, to specify all vowels, use [aeiou]. To specify all punctuation and decimal digit characters, use [\p{P}\d].
[^character_group]	Matches any character not in the specified character group. For example, to specify all consonants, use [^aeiou]. To specify all characters except punctuation and decimal digit characters, use [^\p{P}\d].
[firstCharacter-lastCharacter]	Matches any character in a range of characters. For example, to specify the range of decimal digits from '0' through '9', the range of lowercase letters from 'a' through 'f', and the range of uppercase letters from 'A' through 'F', use [0-9a-fA-F].
.	Matches any character except \n. If modified by the Singleline option, a period matches any character.
\p{name}	Matches any character in the Unicode general category or named block specified by name (for example, Ll, Nd, Z, IsGreek, and IsBoxDrawing).
\P{name}	Matches any character not in Unicode general category or specified named block
\w	Matches any word character. Equivalent to the Unicode general categories [\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]. If ECMAScript-compliant behavior is specified with the ECMAScript option, \w is equivalent to [a-zA-Z_0-9].
\W	Matches any nonword character. Equivalent to the Unicode general categories [^\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]. If ECMAScript-compliant behavior is specified with the ECMAScript option, \W is equivalent to [^a-zA-Z_0-9].
\s	Matches any white-space character. Equivalent to the escape sequences and Unicode general categories [\f\n\r\t\v\x85\p{Z}]. If ECMAScript-compliant behavior is specified with the ECMAScript option, \s is equivalent to [\f\n\r\t\v].

\S	Matches any non-white-space character. Equivalent to the escape sequences and Unicode general categories [^\f\n\r\t\v\x85\p{Z}]. If ECMAScript-compliant behavior is specified with the ECMAScript option, \S is equivalent to [^\f\n\r\t\v].
\d	Matches any decimal digit. Equivalent to \p{Nd} for Unicode and [0-9] for non-Unicode, ECMAScript behavior.
\D	Matches any nondigit character. Equivalent to \P{Nd} for Unicode and [^0-9] for non-Unicode, ECMAScript behavior.

Suppose you want to identify white space in a file. To do this, you can use the match pattern \s which is listed in Table 2-5 as a character pattern. The ability to find white space in a text file is quite useful, because for many items, the end of line separator is just white space. To illustrate working with white space, examine the following RegWhiteSpace.ps1 script.

The first line of the script includes a line of text to use for testing against. The pattern comes from Table 2-5 and is a simple *\s*, which tells the regular expression you want to match on white space. Then use the *$matches* variable to hold the *match* object returned by the *match static* method of the regex type accelerator.

After printing out the results of the match, move to phase two, which is to replace, using the same pattern. To do this, feed the pattern to the *replace* method along with the variable containing the unadulterated text message. Go ahead and print the value of *$strReplace* that now contains the modified object.

RegWhiteSpace.ps1

```
$strText = "a nice line of text. We will search for an expression"

$Pattern = "\s"

$matches = [regex]::match($strText, $pattern)


"Result of using the match method, we get the following:"

$matches


$strReplace = [regex]::replace($strText, $pattern, "_")

"Now we will replace, using the same pattern. We will use

an underscore to replace the space between words:"


$strReplace
```

Using Command-Line Arguments

Modifying a script at run time is an important time-saving, labor-saving, and flexibility-preserving technique. In many companies, first-level support is given the ability to run scripts but not to create scripts. The first-level support personnel do not have access to script editors, nor are they expected to know how to modify a script at design time. The

solution is to use command-line arguments that modify the behavior of the script. In this manner, the scripts become almost like custom-written utilities that are edited by the user, rather than components that are modified via a series of switches and parameters. An example of this technique is shown here in the ArgsShare.ps1 script.

The ArgsShare.ps1 script defines a simple function that is used to perform the WMI query. It takes a single argument from the command line when the script is run. This will determine the kind of shares that are returned.

An *if … else* statement is used to determine if a command-line argument is present. If it is not present, then a friendly help message is displayed that suggests running help for the script. In reality, anything that is not a recognized as a valid argument will result in displaying the help string. The help message suggests the common *question mark* switch.

Once it is determined a valid command-line argument is present, the switch statement will assign the appropriate value to the *$strShare* variable, and will then call the WMI function. This procedure allows a user to type in a simple noun such as: *admin*, *print, file, ipc,* or *all* and generate the appropriate WMI query. However, WMI expects a valid share type integer. By using switch in this way, you generate the appropriate WMI query based upon input received from the command line. If an unexpected command-line argument is supplied, the default switch is used; this simply prints out the help message. You can change this to perform an *all* type of query or some other default WMI query, if desired. You can even paste the your default WMI query into the if(!args) statement and allow the default query to run when there is no argument present. This mimics the behavior of some Windows command-line utilities. The ArgsShare.ps1 script is shown here.

ArgsShare.ps1

```
Function FunWMI($strShare)
 {
  Get-WmiObject win32_share -Filter "type = $strShare"
 }


if(!$args)
{ "you must supply an argument. Try ArgsShare.ps1 ?"}
ELSE
{
$strShare = $args
switch ($strShare)
  {
   "admin" { $strShare = 2147483648 ; funwmi($strShare) }
   "print" { $strShare = 2147483649 ; funwmi($strShare) }
   "file"  { $strShare = 0 ; funwmi($strShare) }
   "ipc"   { $strShare = 2147483651 ; funwmi($strShare) }
   "all"   { Get-WmiObject win32_share }
```

```
   Default { Write-Host "You must supply either: admin, print, file, ipc, or all `n
             Example: > ArgsShare.ps1 admin" }
  }
}
```

Summary

In this chapter we first examined the scripting policy provided by Windows PowerShell. We looked at the steps involved in configuring Windows PowerShell for scripting use. We examined the various flow control statements, and examined scripts that use flow control for advanced scripting needs. We looked at implementing decision making in Windows PowerShell and saw how encapsulated logic can vastly simplify network administration tasks by acting upon routine events when they are presented to the script. Finally, we explored the use of regular expressions to provide advanced pattern-matching capabilities to both scripts and cmdlets.