**External Articles/Sources:**

**Unix:**

Unix Shell shortcuts:

In Vim, :! Lets you run unix commands w/o leaving Vim

:!! Lets you run the last command

. *scriptname* means the script is run in current shell (now in global scope)

exec *scriptname* closes out existing shell and opens a new one just for *scriptname*

In UNIX, need to list interpreter at top (e.g. #!bin/sh)

To turn into CGI script need interpreter then:

echo Content-type: text/plain (only an example, content type may vary)

w/ CGI script only output, no stdin

Grep:

Syntax: Grep –options regexp filepath

-i (ignore case)

-v (doesn’t display line matching)

-n (displays line numbers)

-c (displays count of occurrences)

-l (displays list of filenames only)

-e exp (displays exp with this option. Can be used multiple times and if exp starts with “-“)

-x (matches pattern w/ entire line)

-f file (takes pattern from file, one per line)

Use case: command output | grep (isolates only results that match that regex)

Awk (interactive):

Syntax: awk –options selectioncriteria {action} file

Sed (non-interactive):

Syntax: sed address action file

-n suppress default printing when using p

-e precedes each instruction when using multiple instructions

-f flname takes instructions from file *flname*

i (insert)

a (append)

c (change)

d (delete line)

p (prints line to stdout)

q (quits after reading up to addressed line)

r *flname* (places contents of file *flname* after line)

w *flname* (writes addressed lines to file *flname*)

= (prints line number addressed)

s/s1/s2 (replaces first occurrence of expression s1 in all lines with expression s2)

s/s1/s2/g (as above, but replaces all occurrences)

ex:

1,4d (deletes lines 1 to 4)

10q (quits after reading top 10 lines)

3, $p (prints lines 3 to end (-n option required))

$!n (prints all lines except last line (-n option required))

/begin/ ,/end/p (prints line containing begin through line containing end (-n option required))

s/echo/printf/g (replaces all occurrences of echo with printf)

**PowerShell:**

Within double quotes only, Windows PowerShell will look for the $, and will assume that everything after the $, up to the first character that’s illegal in a variable name, is a variable name. The contents of that variable will replace the variable name and the $:

Windows PowerShell will always interpret zero as False, and any nonzero value as True.

Windows PowerShell also recognizes the built-in variables $True and $False as representing those Boolean values.

> creates a new file

>> appends to a file

2> Redirects Std Err

2>&1 Redirects Std Output AND Std Err

Redirect unneeded output to /dev/null (basically a blackhole)

Run program asynchronously with & at EOL

Ctrl-D = EOF character

Special Characters in UNIX:

&

\*

?

[]

<>

|

()

`

#

$

^

=

‘

“

{}

;

\

Cannot use / in filenames, and can only use \ when escaped (for obvious reasons)

To escape:

Prefix special char with \

Surround char with “” – doesn’t work for all (notably $)

Surround char with ‘’ – works for all except ‘

**Chapter 2: Running Commands**

You must have a space after the command name, so that PowerShell knows that the command name is done, and that whatever comes next is a parameter or value.

A PSDrive is a mapping between the shell and some kind of data store—the filesystem, the registry, or even Active Directory.

You can map new drives by using the New-PSDrive command. Don’t bother doing so now, because it’s something you’ll practice a bit later. Keep in mind that these are PowerShell drives, so you won’t see them in Explorer. They only exist within the shell, and whatever you map will unmap automatically when you close the shell.

* All PowerShell cmdlet names have a strict naming convention. Cmdlet names start with a verb, like Get or Copy, followed by a hyphen, and then a singular noun, such as Item or Content. The list of allowed verbs is quite small—a few dozen or so—and the number of verbs you use on a daily basis will probably number less than a dozen. The idea is that you’ll gradually become used to those verbs and be able to guess new cmdlet names. More on that in a second.
* Cmdlet names tend to be a little generic. Why Move-Item and not Move-File? Keep in mind that the cmdlet has to operate in the registry, environment variables, and other storage systems, as well as the filesystem. Rather than having separate Move-File and Move-RegistryKey cmdlets, PowerShell has a single generic Move-Item.
* Parameter names (-recurse was one example) always start with a dash, and for parameters that accept a value (like the -name DEMO example I showed you earlier), there’s always a space separating the parameter name and the value. Dash, name, space, value. When I teach classes, I make my students repeat that aloud: dash, name, space, value. After that, they never wonder if parameter names should start with a dash or a slash, or if there’s supposed to be an equal sign or colon in between the name and value. It’s “dash, name, space, value,” and never anything else.
* Parameter names are used consistently throughout the shell. If one cmdlet has a -computerName parameter, which is used for specifying a computer name, then most cmdlets that need a computer name will also have a -computerName parameter.
* Both parameter and cmdlet names are intended to be clear and meaningful. When you look at a cmdlet name like Get-Content, you should be pretty clear that it’s getting some kind of content from something. A parameter name like -credential doesn’t leave much to the imagination, either—you should be pretty certain what that parameter is going to do.
* Because “clear” can sometimes mean “lengthy,” Microsoft gives you shortcuts for cmdlet names and parameter names, to save you some typing. Cmdlet names can be given shorter aliases, as we’ve already discussed, and parameter names don’t need to be typed in their entirety. For example, if -recurse is the only parameter of Get-ChildItem that starts with the letter *r*, then you only have to type -r and PowerShell will know what you mean. If a cmdlet has both a -computerName and -credential parameter, typing -comp will probably be enough for PowerShell to figure out that you mean the -computerName parameter.

cmdlet nouns are singular, and never plural

| **Table 2.4. Introducing some new cmdlets** | |
| --- | --- |
| **Task** | **Cmdlet** |
| Display a list of services | Get-Service |
| Display a list of running processes | Get-Process |
| Display the contents of an event log | Get-EventLog |
| Create a new service | New-Service |
| Retrieve Exchange mailboxes | Get-Mailbox |
| Create a new Exchange mailbox | New-Mailbox |

**Chapter 3: The help system**

PowerShell’s help lists optional parameters in square brackets.

Help = Get-Help |more

There are two ways to identify a positional parameter. The first way is right in the syntax summary: the parameter name—just the name—will be surrounded by those square brackets.

Use –full parameter to get more details

For now, you can use single or double quotation marks interchangeably, but it’s best to stick with single quotes.

string[] indicates that the parameter can accept an *array*, or *collection*, or *list* of strings. In these cases, it’s always legal to provide a single value:

Get-EventLog Security -computer Server-R2

But it’s also legal to specify multiple values. A simple way to do so is to provide a comma-separated list. PowerShell treats all comma-separated lists as arrays of values:

Get-EventLog Security -computer Server-R2,DC4,Files02

Another way to provide a list of values is to type them into a text file, with one value per line. Here’s an example:

Server-R2

Files02

Files03

DC04

DC03

Then, you can use the Get-Content cmdlet to read the contents of that file.

By enclosing a command in parentheses, you force that command to execute first:

Get-EventLog Application -computer (Get-Content names.txt)

There’s one more way that you can specify multiple values for a parameter, provided it’s a mandatory parameter: don’t specify the parameter at all. As with all mandatory parameters, PowerShell will prompt you for the parameter value. For parameters that accept multiple values, you can type the first value and press Return. PowerShell will then prompt for a second parameter, which you can type and finish by hitting Return. Keep doing that until you’re finished, and press Return on a blank prompt to let PowerShell know that you’re finished. As always, you can press Ctrl-C to abort the command if you don’t want to be prompted for entries.

Use the –example parameter to get a list of examples.

**Chapter 4: The pipeline**

-whatif parameter lets you preview dangerous actions w/o executing them

**Chapter 5: Adding commands**

If there is a set of addins/snapins/etc. you want to import, put it in a ps1 file to load automatically on start. For more info ‘help \*profile\*’

Third party snapins *should* follow the same syntax for help, and provide similarly formatted info.

Why does this not work?

Create a shortcut with the following target:

%windir%\system32\WindowsPowerShell\v1.0\powershell.exe -noexit -psconsolefile c:\myshell.psc

Why can I not edit Default Domain Policy?

**Chapter 6: Objects**

all of the properties, methods, and other things attached to an object are collectively called its *members*

Probably 90 percent of what you do in PowerShell will involve properties.

For some reason Start-Process DOES NOT require –name tag, but Stop-Process DOES require –name tag

The PowerShell pipeline always contains objects, right until the last command has been executed.

* Remember that the PowerShell help files don’t contain information on objects’ properties. You’ll need to pipe the objects to Gm (Get-Member) to see a list of properties.
* Remember that you can add Gm to the end of any pipeline that normally produces results. A command line like Get-Process -name Notepad | Stop-Process doesn’t normally produce results, so tacking | Gm onto the end won’t produce anything either.
* Start paying attention to neat typing. Put a space on either side of every pipeline character, so that your command lines read like Get-Process | Gm and not Get-Process|Gm. That spacebar key is extra-large for a reason—use it!
* Always remember that the pipeline can contain different types of objects at each step. Think about what type of object is in the pipeline, and focus on what the next command will do to that *type* of object

**Chapter 7: Pipeline Again**

In order to “pipe in” a listed output, the command you’re piping to must be able to accept that input type “ByValue”. (ByPropertyName works a little differently.)

You can get around the above by passing the command in parentheses as a parameter to the function you want to “pipe to”.

In PowerShell, run notepad c:\users.csv. If the file doesn’t exist, Notepad will offer to create it, and it won’t tack on the .txt filename extension.

Useful syntax for reformatting CSV files to match PS paramaters:

Import-CSV c:\users2.csv | Select-Object \*,@{l='samAccountName';e={$\_.LoginName}}, @{l='Name';e={$\_.LoginName}}, @{l='GivenName';e={$\_.FirstName}}, @{l='Surname';e={$\_.LastName}}

placeholder: $\_

When the command runs, PowerShell will fill in this placeholder with the objects that were piped into the next command.

After the underscore, typing a period tells the shell that I don’t want to refer to the entire row, but to a single object instead.

**Chapter 8: Formatting**

Your Format- cmdlet should be the last thing on the command line, with Out-File or Out-Printer as the only real exceptions.

Out-GridView does not accept formatting

Format-Custom is really powerful and useful if trying to combine different data types, but requires a bit of research.

**Chapter 9: Filtering**

* -eq—Equality, as in 5 -eq 5 (which is True) or "hello" -eq "help" (which is False)
* -ne—Not equal to, as in 10 -ne 5 (which is True) or "help" -ne "help" (which is False, because they are, in fact, equal, and we were testing to see if they were inequal)
* -ge and -le—Greater than or equal to, and less than or equal to, as in 10 -ge 5 (True) or Get-Date -le '2012-12-02' (which will depend on when you run this, and shows how dates can be compared in this fashion)
* -gt and -lt—Greater than and less than, as in 10 -lt 10 (False) or 100 -gt 10 (True)
* (5 -gt 10) -and (10 -lt 100) is False, because one or both subexpressions were False.
* (5 -gt 10) -or (10 -lt 100) is True, because at least one subexpression was True.

Windows PowerShell defines $False and $True to represent the False and True Boolean values.

You’ll sometimes see the -not operator abbreviated as an exclamation mark (!).

* -like accepts \* as a wildcard, so you can compare to see if "Hello" -like "\*ll\*" (that would be True). -notlike is the reverse, and both are case-insensitive; use -clike and -cnotlike for case-sensitive comparisons.
* -match makes a comparison between a string of text and a regular expression pattern. -notmatch is its logical opposite, and as you might expect, -cmatch and -cnotmatch provide case-sensitive versions. Regular expressions are beyond the scope of what we’ll cover in this book.

## Above and beyond

If a cmdlet doesn’t use the preceding PowerShell-style comparison operators, it probably uses the more traditional, programming language-style comparison operators that you might remember from high school or college (or even your daily work!):

* = equality
* <> inequality
* <= less than or equal to
* >= greater than or equal to
* > greater than
* < less than

If Boolean operators are supported, they’re usually the words AND and OR; some cmdlets may support operators such as LIKE as well. You’ll find support for all of these operators in the -filter parameter of Get-WmiObject, for example, and I’ll repeat this list when we discuss that cmdlet in [**chapter 11**](http://my.safaribooksonline.com/9781617290213/ch11#ch11).

Every cmdlet’s designers get to pick how (and if) they’ll handle filtering; you can often get examples of what they decided to do by reviewing the cmdlet’s full help, including the usage examples near the end of the help file.

**Chapter 10: Remoting**

Enable-PSRemoting gets the computer all set.

* Even if you have a PowerShell profile script on the remote computer, it won’t run when you connect using remoting. We haven’t fully covered profile scripts yet (they’re in [chapter 24](http://my.safaribooksonline.com/9781617290213/ch24#ch24)), but suffice to say that they’re a batch of commands that run automatically each time you open the shell. Folks use them to automatically load shell extensions and modules and so forth. That doesn’t happen when you remote into a computer, so be aware of that.
* You’re still restricted by the remote computer’s execution policy. Let’s say your local computer’s policy is set to RemoteSigned, so that you can run local, unsigned scripts. That’s great, but if the remote computer’s policy is set to the default, Restricted, it won’t be running any scripts for you when you’re remoting into it.
* Always complete as much of your processing on the remote computer as possible.
* Remoting only works, by default, with the remote computer’s real computer name. You can’t use DNS aliases or IP addresses.
* When you invoke a command, you’re asking the remote computer to launch PowerShell, run your command, and then close PowerShell. The next command you invoke on that same remote computer will be starting from scratch—anything that was run in the first invocation will no longer be in effect. If you need to run a whole series of related commands, put them all into the same invocation.
* If you’re using a local firewall product other than the Windows Firewall, Enable-PSRemoting won’t set up the necessary firewall exceptions. You’ll need to do so manually. If your remoting connection will need to traverse a regular firewall, such as one implemented on a router or proxy, then it’ll also need a manually entered exception for the remoting traffic.
* Don’t forget that any settings in a Group Policy object (GPO) override anything you configure locally. I’ve seen administrators struggle for hours to get remoting working, only to finally discover that a GPO was overriding everything they did. In some cases, that GPO was put into place a long time ago by a well-meaning colleague, who had long since forgotten it was there. Don’t assume that there’s no GPO affecting you; check and see for sure.

**Chapter 11: WMI**

If you find yourself in possession of a corrupted repository, check out the “Repairing and re-registering the WMI” article on Ramesh Srinivasan’s Troubleshooting Windows blog (<http://windowsxp.mvps.org/repairwmi.htm>).

For WMI GUI: ([**http://www.primaltools.com/downloads/communitytools/**](http://www.primaltools.com/downloads/communitytools/))

There are some things you should notice about wgmi’s –filter parameter:

* The filter criteria is usually enclosed in double quotation marks.
* The filter comparison operators aren’t the normal PowerShell -eq or -like operators. Instead, WMI uses more traditional, programming-like operators, such as =, >, <, <=, >=, and <>. You can use the keyword LIKE as an operator, and when you do your comparison value can use % as a character wildcard, as in "NAME LIKE '%administrator%'".
* Any string comparison values are enclosed in single quotation marks, which is why the outermost quotes that contain the entire filter expression must be double quotes.
* Backslashes are escape characters for WMI, so when you need to use a literal backslash, as in this example, you have to use two backslashes.
* The output of Gwmi always includes a number of system properties. These are often suppressed by PowerShell’s default display configuration, but they’ll be displayed if you’re deliberately listing all properties or if the class doesn’t have a default. System property names start with a double underscore. Here are two particularly useful ones:
  + \_\_SERVER contains the name of the computer that the instance was retrieved from. This can be useful when retrieving WMI information from multiple computers at once.
  + \_\_PATH is an absolute reference to the instance itself, and it can be used to requery the instance if necessary.

**Chapter 12: Multitasking/Background Jobs**

To retrieve the results from a job, use Receive-Job. Before you run this, you need to know a few things:

* You have to specify the job you want to receive results from. You can do this by job ID, job name, or by getting jobs with Get-Job and piping them to Receive-Job.
* If you receive the results of the parent job, those results will include all output from all child jobs. Alternatively, you can choose to just get the results from one or more child jobs.
* Normally, receiving the results from a job clears them out of the job output cache, so you can’t get them a second time. Specify -keep to keep a copy of the results in memory. Or, you can output the results to CliXML if you want to retain a copy to work with.
* The job results may be deserialized objects, which you learned about in [chapter 10](http://my.safaribooksonline.com/9781617290213/ch10#ch10). That means they’re a snapshot from the point in time when they were generated, and they may not have any methods that you can execute. But you can pipe the job results directly to cmdlets such as Sort-Object, Format-List, Export-CSV, ConvertTo-HTML, Out-File, and so on, if desired.
* Remove-Job— This deletes a job, and any output still cached with it, from memory.
* Stop-Job— If a job seems to be stuck, this command will terminate it. You’ll still be able to receive whatever results were generated to that point.
* Wait-Job— This is useful if a script is going to start a job and you want the script to continue only when the job is done. This command forces the shell to stop and wait until the job is completed, and then allows the shell to continue.

-command = -scriptblock

**Chapter 13: Working with Bunches of Objects**

“If you can get to something by using Get-WmiObject, then Invoke-WmiObject can execute its methods.”

% is a super-short alias for ForEach-Object

* Native cmdlets’ filtering criteria usually use \* as a wildcard character, where WMI filtering uses the percent sign (%)—don’t confuse that percent sign for the ForEach-Object alias! This percent sign is enclosed within the value of Get-WmiObject's -filter parameter, and it isn’t an alias.
* Native objects often have similar capabilities to WMI ones, but the syntax may differ. Here, the ServiceController objects produced by Get-Service have a Stop() method; when I access those same services through the WMI Win32\_Service class, the method name becomes StopService().
* Native filtering often uses native comparison operators, such as -eq; WMI uses programming-style operators such as = or LIKE.
* If you retrieved something by using Get-WmiObject, you’ll take action on that something by using a WMI method. You can execute the method by using Invoke-WmiMethod or the ForEach-Objectapproach.
* If you retrieved something by using an approach other than Get-WmiObject, you’ll use a native cmdlet to take action against that something. Or, if whatever you retrieved has a method but no supporting cmdlet, you might use the ForEach-Object approach to execute that method.

WMI methods aren’t documented in PowerShell’s built-in help system; you’ll need to use a search engine (usually searching on the WMI class name) to locate WMI method instructions and examples.

To find the documentation for these, focus on the TypeName, which in this case is System.ServiceProcess.ServiceController. Search for that complete type name in a search engine, and you’ll usually come across the official developer documentation for that type, which will lead to the documentation for whatever specific method you’re after.

**Chapter 14: Security**

**Chapter 15: Variables**

‘’ is treated as literal

“” not-so-much (variable exceptions, etc.)

`n = newline

`t = tab

$array.count = len(array) in python

$str.length seems to be only for strings

$str.replace(‘old’,’new’)

Cannot call member methods for entire array, must specify all members

Example of executing function on all members of an array:

$computers = $computers | ForEach-Object { $\_.ToLower() }

Example of getting properties for all members of an array:

$computers | select-object length

* [int]— Integer numbers
* [single] and [double]— Single-precision and double-precision floating numbers (numbers with a decimal portion)
* [string]— A string of characters
* [char]— Exactly one character (as in, [char]$c = 'X')
* [xml]— An XML document; whatever string you assign to this will be parsed to make sure it contains valid XML markup (for example, [xml]$doc = Get-Content MyXML.xml)
* [adsi]— An Active Directory Services Interface (ADSI) query; the shell will execute the query and place the resulting object or objects into the variable (such as [adsi]$user = "WinNT:\\MYDOMAIN\Administrator,user")

Helpful variable commands:

* New-Variable
* Set-Variable
* Remove-Variable
* Get-Variable
* Clear-Variable

The shell has two parsing rules that let it capture the variable name:

* If the character immediately after the dollar sign is a letter, number, or underscore, the variable name consists of all the characters following the dollar sign, up to the next white space (which might be a space, tab, or carriage return).
* If the character immediately after the dollar sign is an opening curly brace, {, the variable name is everything after that curly brace up to, but not including, the closing curly brace, }.

**Chapter 16: Input & Output**

When the configuration variable is set to Continue, the commands I’m about to show you do indeed produce output. When the configuration variable is set to SilentlyContinue, the associated output command produces nothing.

| Table 16.1. Alternative output cmdlets | | |
| --- | --- | --- |
| **Cmdlet** | **Purpose** | **Configuration variable** |
| Write-Warning | Displays warning text, in yellow by default and preceded by the label “WARNING:” | $WarningPreference(Continue by default) |
| Write-Verbose | Displays additional informative text, in yellow by default and preceded by the label “VERBOSE:” | $VerbosePreference(SilentlyContinue by default) |
| Write-Debug | Displays debugging text, in yellow by default and preceded by the label “DEBUG:” | $DebugPreference(SilentlyContinue by default) |
| Write-Error | Produces an error message | $ErrorActionPreference(Continue by default) |

**Chapter 17: Scripting Intro**

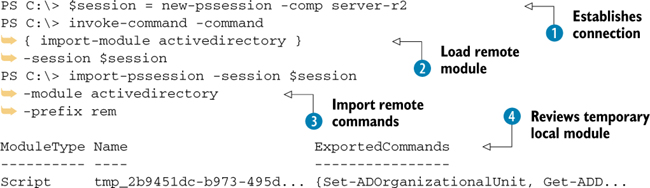
I formatted my command so that each physical line ends in either a pipe character or a comma. By doing so, I’m forcing the shell to recognize these multiple lines as a single, one-line command.

If you try to access a scoped element, PowerShell sees if it exists within the current scope. If it doesn’t, PowerShell sees if it exists in the current scope’s parent. It continues going up the relationship tree until it gets to the global scope.

when a scope defines a variable, alias, or function, that scope loses access to any variables, aliases, or functions having the same name in a parent scope

**Chapter 18: Sessions and Making Remoting Easier**

Remoting works over a single, predefined port; WMI doesn’t. Remoting is therefore easier to use with computers that are firewalled, because it’s easier to make the necessary firewall exceptions. Microsoft’s Windows Firewall provides a specific exception for WMI that includes the stateful inspection necessary to make WMI’s random port selection (called *endpoint mapping*) work properly, but it can be difficult to manage with some third-party firewall products. With remoting, it’s an easy, single port.

****

**Chapter 19: Command -> Script -> Function**

The return keyword places a single object (such as an integer, in this example) into the pipeline, and then immediately exits the function. Any code following the return keyword won’t ever execute.

* *Break tasks down—* We separated how we get the computer name from the actual working code. All of the working code went into a single function, because we want a single unified piece of output.
* *Output objects—* Always have functions return either a single value (using the return keyword), or output objects. By outputting objects, you can pipe the function’s output to many other cmdlets to format, convert, filter, sort, and so forth.

. .\script.ps1 (allows script to run without snapping back to global scope [thus losing all info]) 🡨 messy, because it’s hard to remove them

**Chapter 20: Logic and Loops**

This chapter will introduce you to all but one of PowerShell’s formal scripting constructs. The missing construct is the Do loop, which also uses the keywords Whileand Until in certain scenarios. You can learn more about them in PowerShell’s help: run help about\* to get a list of help topics, and look for Do, While, and Until.

* As with most of PowerShell, the If keyword isn’t case-sensitive. You can use if or IF or even iF.
* The parentheses contain an expression of some kind. This has to evaluate to either True or False (or, to use the PowerShell values, $True or $False).
* After the parentheses, you open the construct by using a curly brace. You complete the construct with a closing brace.
* Most people indent the commands within the construct, so that it’s easier to visually distinguish the commands that are inside the construct.

The Switch construct acts as a specialized kind of logical comparison. You start with a single variable or property, and you ask the shell to compare its contents to a wide range of possible values. The shell will execute a block of commands for each match that it finds. The Default block will execute only if none of the prior blocks have executed.

You might only want the first matching condition to execute. In those cases, use the Break keyword within one of the conditional blocks. Break exits the entire construct immediately (it will exit anything except an If construct), preventing further potential matches from being considered.

The Switch construct has a few other tricks it can perform, including evaluating regular expressions.

For ($i = 0; $i -lt 10; $i++) {

Write-Host $i

}

$services = Get-Service

ForEach ($service in $services) {

Write-Host $service.Name

}

^^==^^

Get-Service | ForEach-Object { Write-Host $\_.Name }

^^==^^

Get-Service | Select Name

**Chapter 21: Making your Own Cmdlets and Modules**

A filtering function has these distinguishing characteristics:

* You can accept one kind of information through the pipeline. This might be computer names, processes, or any other single kind of information.
* Whatever you accept through the pipeline can come as a single object, or multiple objects can be piped in. You’ll write one (or many) commands that execute against each piped-in object, no matter how many there are.
* You can designate additional parameters for other input elements. The values provided to these parameters will be used for each execution of your commands.

A *script cmdlet* by the PowerShell community is formally called an *advanced function* in PowerShell’s documentation.

The values for ConfirmImpact are Low, Medium, and High, and it’s entirely up to your discretion which one you use. I tend to think of these settings as, “what is the likelihood of someone getting fired if they do this accidentally?”

**Chapter 22: Dealing with Errors**

The shell’s default error-handling behavior is defined by a built-in variable called $ErrorActionPreference. When you open a new shell session, this variable is set toContinue. Its possible values, and their functions, are as follows:

* SilentlyContinue— For nonterminating problems, don’t display an error message—just keep going.
* Continue— For nonterminating problems, display an error message and keep going.
* Inquire— For nonterminating problems, ask what to do using an interactive prompt to which the user must respond.
* Stop— Stop executing and throw an exception.
* The -ErrorAction parameter, or its alias -EA, is one of the common parameters supported by every cmdlet that runs in PowerShell. Using this parameter, you can override the $ErrorActionPreference setting for just that cmdlet.

A Trap construct is defined before you anticipate the error occurring, meaning that the construct needs to appear in your script before the command that might generate the error you want to trap. It’s possible to declare multiple Trap constructs, with each one trapping a different kind of error.

The Trap construct ends with the keyword continue, which tells the shell to resume execution within the same scope.

The other way to end a Trap is by using the keyword break. That exits the current scope and passes the exception up to the parent scope.

You *always* have to have the Try portion, and you *can* have either a Catch, a Finally, or both a Catch and a Finally. As I mentioned earlier, you can also include multiple Catch blocks if you want to handle different exceptions in different ways.

Use Trap constructs as a high-level catch for unanticipated errors, and stick with Try for specific errors on specific commands.

$error[0] will be the exception that occurred most recently (echo $? equivalent?)

All cmdlets support both -EA and -EV, although cmdlets’ help files only list the common parameters set name.

Technically, the $ isn’t a part of the variable’s name. The $tells the shell that the following characters will be a variable name, up to the next white space. If the variable name is enclosed in curly braces, the name may contain spaces.

When the shell needs a variable name from you, that name never includes the $. When you want to get to the information that’s inside a variable, you specify the $ before the variable name.

**Chapter 23: Debugging**

You can help yourself avoid syntax errors by using a quality third-party PowerShell console or editor, such as SAPIEN PrimalScript ([**www.primalscript.com**](http://www.primalscript.com/)), Idera PowerShell Plus ([**www.idera.com**](http://www.idera.com/)), or PowerGUI ([**www.powergui.org**](http://www.powergui.org/); there are both free and commercial versions). These products all include a few common features:

* *Code hinting—* Reminds you of the available parameters and helps type them for you—saving time, and helping to protect against typos.
* *Syntax highlighting—* Colors valid syntax elements in a specific way, with invalid syntax often getting a different color. That helps to visually alert you to a potential problem.
* *Live syntax checking—* Works a bit like the spell-check feature in Microsoft Word: the product puts a red underline underneath bits it doesn’t think are correct, such as invalid parameter names.

Here’s a checklist:

* Make sure you typed the cmdlet name correctly. If you used an alias, make sure it’s typed correctly, and that it points to the cmdlet you think it does. You can run Get-Alias alias (insert your alias name for alias) to double-check which cmdlet an alias points to.
* Make sure parameter names are preceded by a dash and are followed by a space. Make sure you’re using the correct parameter name (read the help!), and if you’re abbreviating the parameter name, make sure you’re providing enough characters to uniquely identify that parameter.
* Most of PowerShell’s punctuation comes in pairs: single quotes, double quotes, square brackets, curly braces, and parentheses are all good examples. Make sure that you end every set that you start, and that you properly nest them. Improper nesting, like ({this)}, means you’re ending a pair before ending the pair it encloses. In that example, I closed the parentheses before the curly braces, which is the opposite of the correct order.
* Watch your spaces. In PowerShell, spaces are special characters that indicate a separation between command elements. PowerShell isn’t that case-sensitive (meaning that upper- and lowercase are usually the same to the shell), but it’s very space-sensitive. There’s a space after a cmdlet name and before any parameters or values. There’s a space in between parameter names and values. There’s a space after one parameter and before the next. Don’t forget those.

TechNet has a nice “Modifying Message Colors” article on the available options ([**http://mng.bz/1037**](http://mng.bz/1037)).

For trace code: Write-Debug "Test message"

But don’t forget: $DebugPreference = "Continue"

Here are my guidelines:

* Whenever I change the contents of a variable, I use Write-Debug to output the variable, just so I can check those contents.
* Whenever I’m going to read the value of a property or a variable, I use Write-Debug to output that property or variable, so that I can see what’s going on inside the script.
* Any time I have a loop or logic construct, I build it in such a way that I get a Write-Debug message no matter how the loop or logic works out. In this example, I added an Else section specifically to have debug output—the Else portion of the construct has no other purpose.

A *breakpoint* is a defined area where a script will pause its execution, allowing you to examine the environment that the script is running within. PowerShell can be configured to break when

* Your script reaches a certain line
* A variable is read and/or changed
* A specific command is executed

In the first instance, you must specify the script file that you’re referring to. In the second and third situations, you can choose to specify a script, and the breakpoint will only be active for that script. If you don’t, the breakpoint will occur globally throughout the shell when that variable is read or written, or that command is executed.

To set a line breakpoint, move your cursor to the desired line and press F9.

Cls to clear shell session output

**Chapter 24: Tips and Tricks**

String variables are typically actually objects <- these objects are where you do string manipulation, not with a straight string like in other languages.

-as [type] converts output to type in []

-is [type] tests if variable type matches type in []

-replace “1”,”2” replaces all instances of 1 with 2 in a string

-join “:” concatenates strings in an array, with delimiter inside “”

-split “:” creates an array from a list with delimiter inside “”

Here are the files that the console host tries to load, and the order in which it tries to load them:

1. $pshome/profile.ps1—This will execute for all users of the computer, no matter which host they’re using (remember that $pshome is predefined within PowerShell and contains the path of the PowerShell installation folder).
2. $pshome/Microsoft.PowerShell\_profile.ps1—This will execute for all users of the computer if they’re using the console host. If they’re using the PowerShell ISE, the $pshome/Microsoft.PowerShellISE\_profile.ps1 script will be executed instead.
3. $home/Documents/WindowsPowerShell/profile.ps1—This will execute only for the current user (because it lives under the user’s home directory), no matter which host they’re using.
4. $home/Documents/WindowsPowerShell/Microsoft.PowerShell\_profile.ps1—This will execute for the current user if they’re using the console host. If they’re using the PowerShell ISE, the $home/Documents/WindowsPowerShell/Microsoft.PowerShellISE\_profile.ps1 script will be executed instead.

Types can include [string], [xml],[int], [single], [double], [datetime]

Some of the more useful String methods include the following:

* IndexOf() tells you the location of a given character within the string.
* PS C:\> "SERVER-R2".IndexOf("-")

6

* Split(), Join(), and Replace() operate similarly to the -split, -join, and -replace operators I described in the previous section. I tend to use the PowerShell operators rather than the String methods.
* ToLower() and ToUpper() convert the case of a string.
* PS C:\> $computername = "SERVER17"
* PS C:\> $computername.tolower()

server17

* Trim() removes white space from both ends of a string; TrimStart() and TrimEnd() remove white space from the beginning or end of a string.
* PS C:\> $username = " Don"
* PS C:\> $username.Trim()

Don

To make WMI date info easier to read:

PS C:\> get-wmiobject win32\_operatingsystem | select BuildNumber,\_\_SERVER,@{

l='LastBootTime';e={$\_.ConvertToDateTime($\_.LastBootupTime)}}

**Chapter 25: Final Words and Practical Help**

* You probably *will* get stuck. I almost always do. Don’t be afraid to ask for help! You can use the community forums at [**www.PoshComm.org**](http://www.poshcomm.org/), or you can log into my own forums at [**http://connect.ConcentratedTech.com**](http://connect.concentratedtech.com/) (if you’re asked where you took a class with me, just answer “Lunches book” and you’ll be allowed in). Manning ([**www.manning-sandbox.com/forum.jspa?forumID=723**](http://www.manning-sandbox.com/forum.jspa?forumID=723)) also has a forum dedicated to this book where you can post questions, and I try to monitor those a couple of times a week.
* Spend some time breaking the task down to its main components. Figure out which parts of the task involve the real functionality, and focus on writing commands that create the desired output.
* Once you have completed the necessary commands, you can worry about writing functions and other structures around those commands.

A lot of experienced PowerShell/AD gurus don’t like the Microsoft cmdlets for a number of reasons, one of which is their inability to access schema extensions, Terminal Services attributes, and so forth.

If you’re a serious AD administrator, you’ll want to check out Quest Software ([**http://quest.com/powershell**](http://quest.com/powershell)) and download their free PowerShell Commands for Active Directory.

Resources:

* [**MoreLunches.com**](http://morelunches.com/)—This should be your first stop, if you haven’t already bookmarked the site. There you’ll find free bonus and companion content for this book, including the lab answers, video demonstrations, bonus articles, and additional recommended resources. You’ll also be able to download the longer code listings for this book, so that you don’t have to type them in manually. Consider bookmarking the site and visiting often to refresh what you’ve learned in this book.
* [**http://WindowsITPro.com/go/DonJonesPowerShell**](http://windowsitpro.com/go/DonJonesPowerShell)—This is a landing page for my online Frequently Asked Questions (FAQ) and blog about Windows PowerShell. You’ll also find bimonthly feature articles. The layout of the page changes from time to time, so if you have trouble finding the blog articles, go directly to the blog index at[**http://www.windowsitpro.com/blogs/PowerShellwithaPurpose.aspx**](http://www.windowsitpro.com/blogs/PowerShellwithaPurpose.aspx). I post a new blog article at least twice weekly, and they’re always either tutorials, tips, or PowerShell-related product reviews.
* [**http://Connect.ConcentratedTech.com**](http://connect.concentratedtech.com/)—This is a private discussion forum for past students—and that now includes you. You’ll need to register for an account, but once you do, you’re welcome to post your PowerShell questions and I’ll do my best to answer. I also monitor a forum hosted by Manning, [**http://www.manning-sandbox.com/forumindex.jspa**](http://www.manning-sandbox.com/forumindex.jspa), if you’d prefer to use that.
* [**http://ShellHub.com**](http://shellhub.com/)—This is a website that I maintain. It’s a handpicked list of other PowerShell-related online resources, including the blogs I read most, third-party PowerShell tools, and more. Pretty much every URL I’ve ever recommended to someone is listed here. In the event that any other URL I give you changes, you can hop on[**ShellHub.com**](http://shellhub.com/) to find an update.

@{label='Column\_or\_Property\_Name';expression={Value\_expression}}