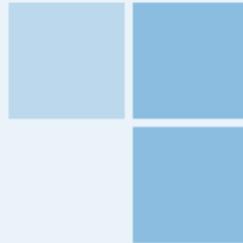


The Big Book of PowerShell Gotchas

by Don Jones

curated by
Mike Shepard



PowerShell.org

The Big Book of PowerShell Gotchas

By Don Jones

Visit PowerShell.org to check for newer editions of this ebook.



Copyright © PowerShell.org, Inc.
All Rights Reserved.

This guide is released under the Creative Commons Attribution-NoDerivs 3.0 Unported License. The authors encourage you to redistribute this file as widely as possible, but ask that you do not modify the document.

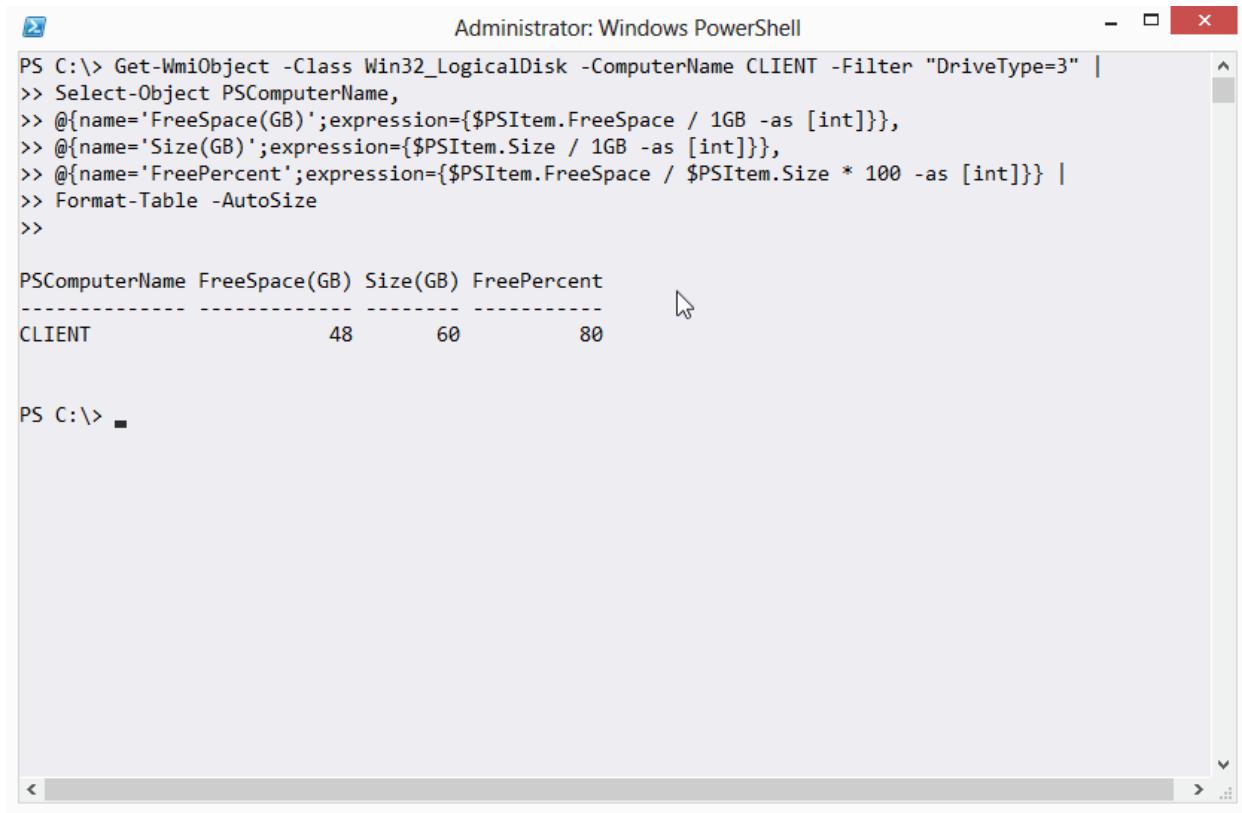
PowerShell.org ebooks are works-in-progress, and many are curated by members of the community. We encourage you to check back for new editions at least twice a year, by visiting PowerShell.org. You may also subscribe to our monthly e-mail TechLetter for notifications of updated ebook editions. Visit PowerShell.org for more information on the newsletter.

Feedback and corrections, as well as questions about this ebook's content, can be posted in the PowerShell Q&A forum on PowerShell.org. Moderators will make every attempt to either address your concern, or to engage the appropriate ebook author.

Format Right.....	4
-Contains isn't -Like.....	7
You Can't Have What You Don't Have	11
-Filter Values Diversity	14
Not Everything Produces Output	15
One HTML Page at a Time, Please	18
[Bloody] {Awful} (Punctuation).....	20
Don't+Concatenate+Strings.....	21
\$ isn't Part of the Variable Name.....	23
Use the Pipeline, not an Array	25
Backtick, Grave Accent, Escape	27
A Crowd isn't an Individual.....	32
These aren't Your Father's Commands	34

Format Right

Everyone runs into this one. Here's how it goes: you start by writing a truly awesome command.



The screenshot shows an Administrator Windows PowerShell window. The command entered is:

```
PS C:\> Get-WmiObject -Class Win32_LogicalDisk -ComputerName CLIENT -Filter "DriveType=3" | >> Select-Object PSComputerName, @{name='FreeSpace(GB)';expression={$PSItem.FreeSpace / 1GB -as [int]}}, @{name='Size(GB)';expression={$PSItem.Size / 1GB -as [int]}}, @{name='FreePercent';expression={$PSItem.FreeSpace / $PSItem.Size * 100 -as [int]}} | >> Format-Table -AutoSize >>
```

The output is a table:

PSComputerName	FreeSpace(GB)	Size(GB)	FreePercent
CLIENT	48	60	80

And you think, “wow, that’d go great in an HTML file.”

A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> Get-WmiObject -Class Win32_LogicalDisk -ComputerName CLIENT -Filter "DriveType=3" | >> Select-Object PSComputerName, >> @{name='FreeSpace(GB)';expression={$PSItem.FreeSpace / 1GB -as [int]}}, >> @{name='Size(GB)';expression={$PSItem.Size / 1GB -as [int]}}, >> @{name='FreePercent';expression={$PSItem.FreeSpace / $PSItem.Size * 100 -as [int]}} | >> Format-Table -AutoSize | >> ConvertTo-HTML | Out-File diskreport.html >> PS C:\>
```

The output of the command is displayed in an embedded browser window titled "HTML TABLE". The content of the table includes:

ClassId2e4f51ef21dd47e99d3c952918aff9cd	pageHeaderEntry	pageFooterEntry	auto
033ecb2bc07a4d43b5ef94ed5a35d280			Microsoft.PowerShell.Command
9e210fe47d09416682b841769c78b8a3			
27c87ef9bbda4f709f6b4002fa4af63c			
4ec4f0187cb04f4cb6973460dfe252df			
cf522b78d86c486691226b40aa69e95c			

Wait... what?!?!

This happens all the time. If you want an easy way to remember what *not* to do, it's this: *Never pipe a Format command to anything else*. That isn't the whole truth, and we'll get to the whole truth in a sec, but if you just want a quick answer, that's it. In the community, we call it the "Format Right" rule, because you have to move your Format command to the right-most end of the command line. That is, the Format command comes *last*, and nothing else comes after it.

The reason is that the Format commands all produce special internal formatting codes, that are really just intended to create an on-screen display. Piping those codes to anything else - ConvertTo-HTML, Export-Csv, whatever - just gets you gibberish output.

In fact, there are actually a few commands that *can* come after a Format command in the pipeline:

Out-Default. This is technically always at the end of the pipeline, although it's invisible. It redirects to Out-Host.

Out-Host also understands the output of Format commands, because Out-Host is how those formatting codes get on the screen in the first place.

Out-Printer understands the formatting codes too, and constructs a printed page that would look exactly like the normal on-screen output.

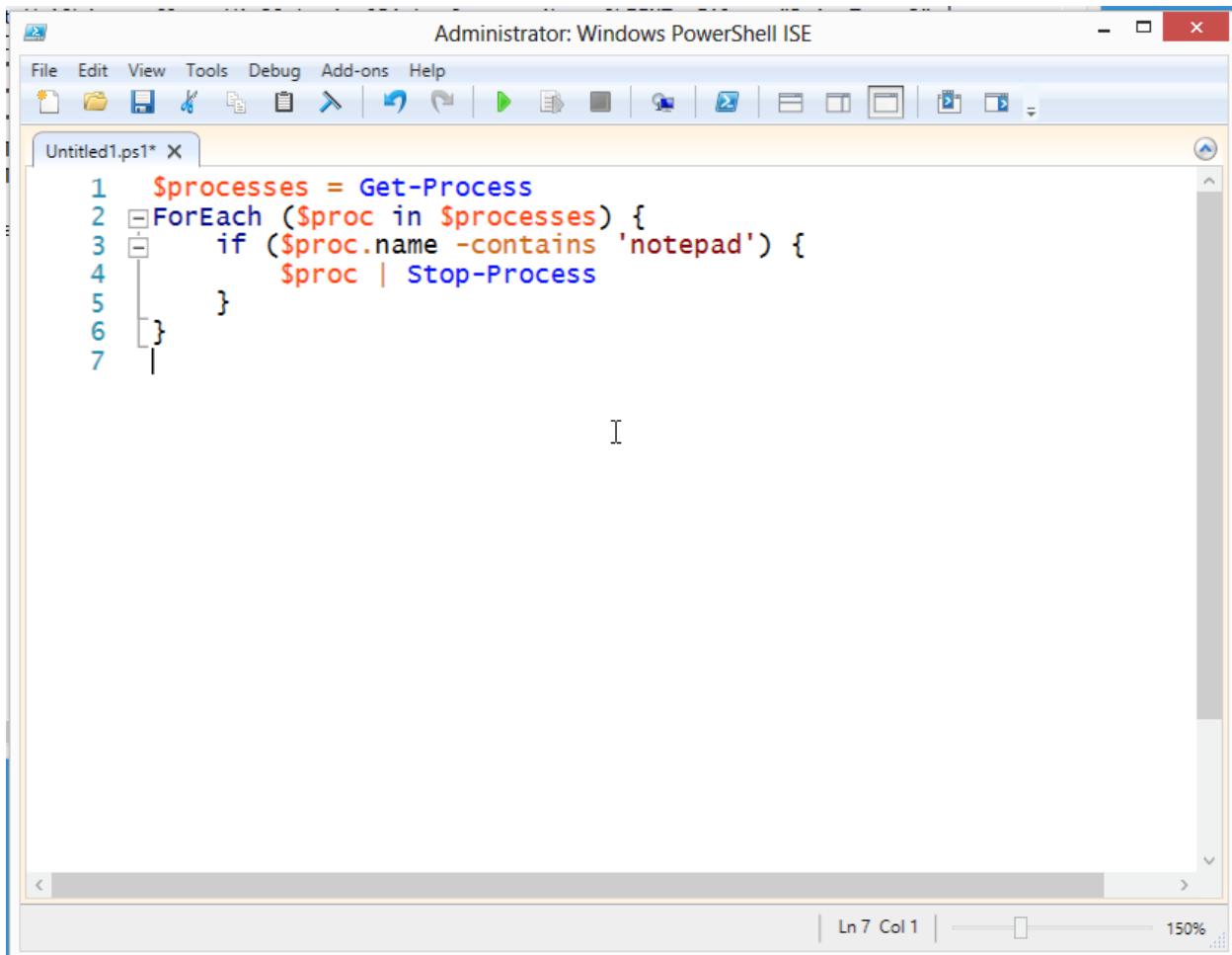
Out-File, like Out-Printer, redirects the on-screen output, but this time to a text file on disk.

Out-String consumes the formatting codes and just outputs a plain string containing the text that would otherwise have appeared on-screen.

Apart from those exceptions - and of them, you'll mainly only ever use Out-File - you can't pipe the output of a Format command to much else and get anything that looks useful.

-Contains isn't -Like

Oh, if I had a nickel for every time I've seen this:



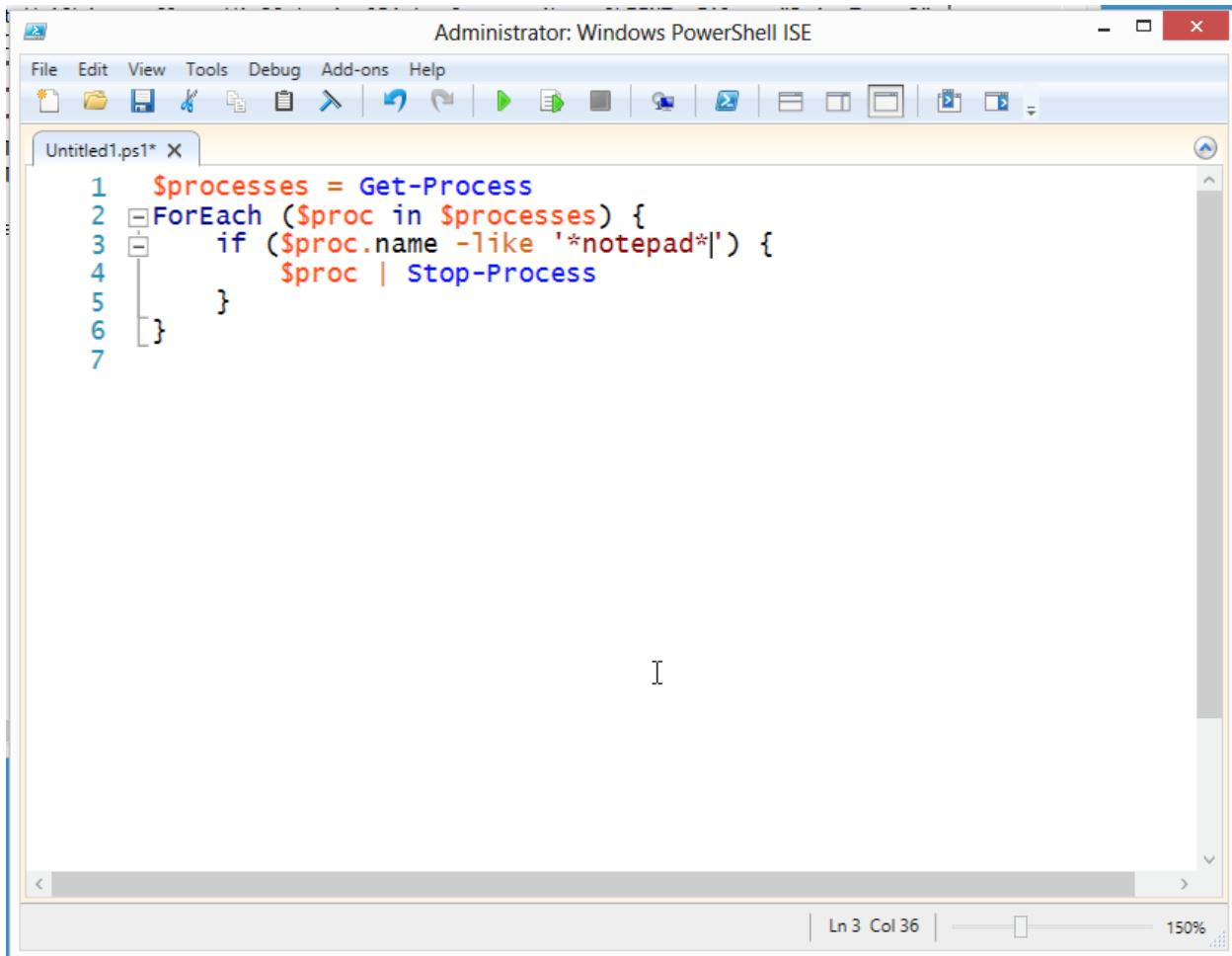
A screenshot of the Windows PowerShell ISE (Integrated Scripting Environment) window. The title bar says "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar has various icons for file operations like Open, Save, Copy, Paste, and Run. A tab labeled "Untitled1.ps1*" is open, containing the following PowerShell script:

```
1 $processes = Get-Process
2 ForEach ($proc in $processes) {
3     if ($proc.name -contains 'notepad') {
4         $proc | Stop-Process
5     }
6 }
7
```

The script uses the `-contains` operator to filter processes named "notepad". The code is highlighted in blue and red, with the error being the misspelling of `-contains` as `-contains`. The ISE shows the error by underlining the misspelled word and displaying a tooltip or error message above it.

I get how this happens. The `-contains` operator *seems* like it should be checking to see if a process' name *contains* the letters "notepad." But that isn't what it does.

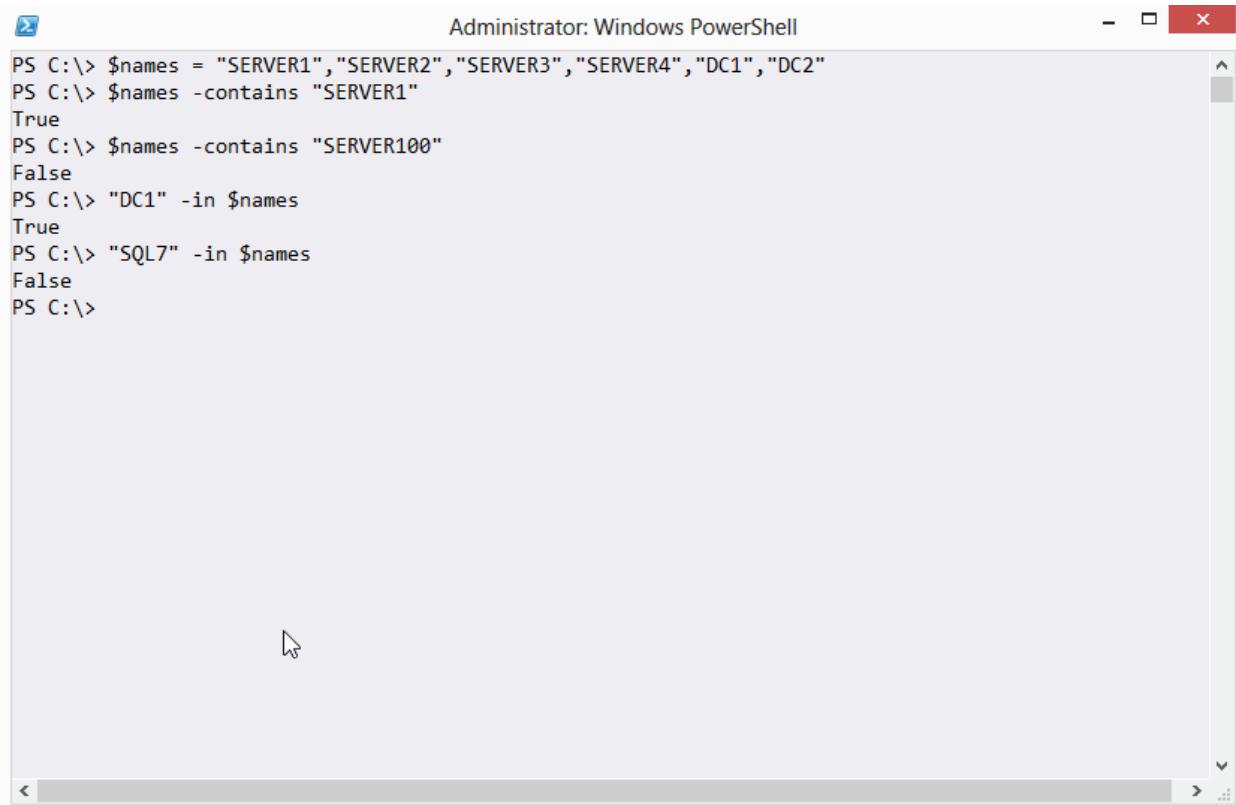
The correct approach is to use the `-like` operator, which in fact *does* do a wildcard string comparison:



```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Untitled1.ps1* X
1 $processes = Get-Process
2 ForEach ($proc in $processes) {
3     if ($proc.name -like '*notepad*') {
4         $proc | Stop-Process
5     }
6 }
7
```

I'll let pass the thought that the *really correct* answer is to just run **Stop-Process -name *notepad***, because I was aiming for a simple example here. But... don't overthink things. Sometimes a script and a ForEach loop isn't the best approach.

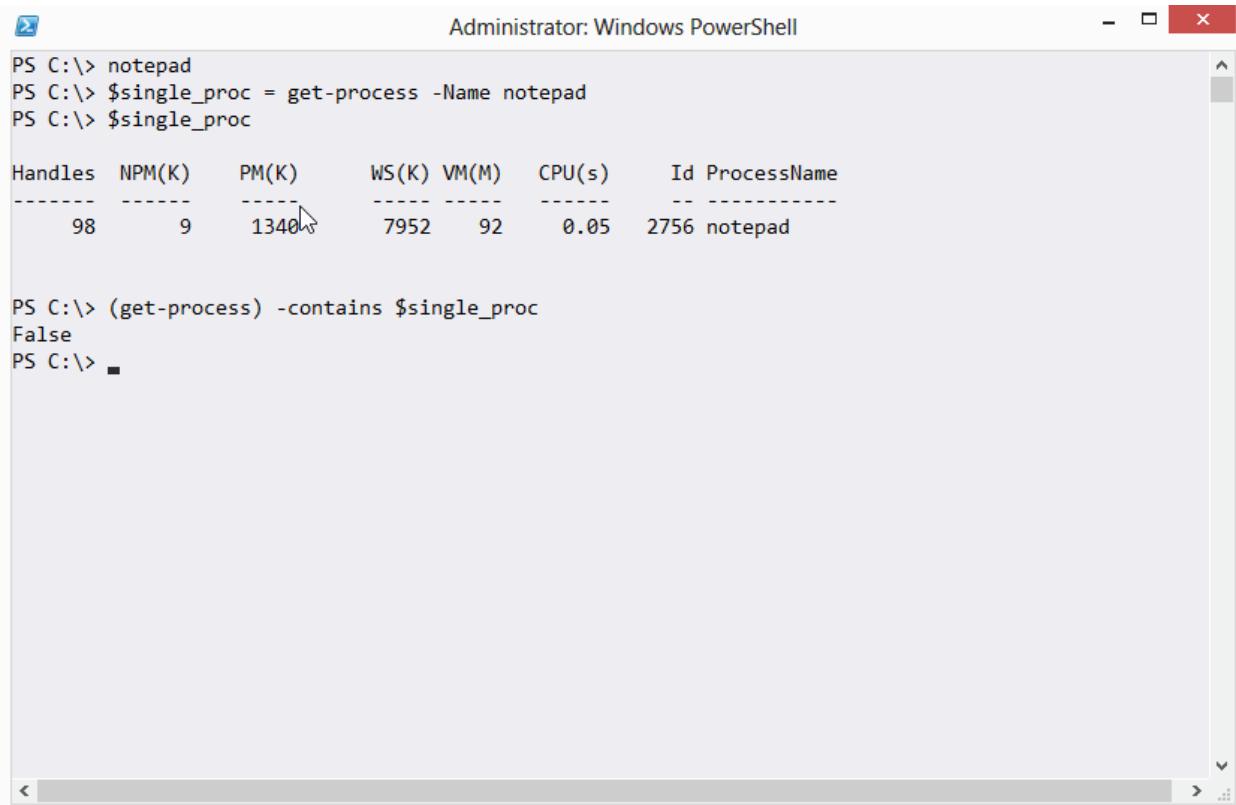
So anyway, what does -contains (and its friend, -notcontains) actually do? They're similar to the -in and -notin operators introduced in PowerShell v3, and *those* operators cause more than a bit of confusion, too. What they do is check to see if a collection of objects contains a given single object. For example:



A screenshot of an "Administrator: Windows PowerShell" window. The window title bar says "Administrator: Windows PowerShell". The content area shows the following PowerShell session:

```
PS C:\> $names = "SERVER1","SERVER2","SERVER3","SERVER4","DC1","DC2"
PS C:\> $names -contains "SERVER1"
True
PS C:\> $names -contains "SERVER100"
False
PS C:\> "DC1" -in $names
True
PS C:\> "SQL7" -in $names
False
PS C:\>
```

In fact, that example is probably the best way to see it work. The trick is that, when you use a complex object instead of a simple value (as I did in that example), `-contains` and `-in` *look at every property of the object* to make a match. If you think about something like a process, they're *always* changing. From moment to moment, a process' CPU and memory, for example, are different.



Administrator: Windows PowerShell

```
PS C:\> notepad
PS C:\> $single_proc = get-process -Name notepad
PS C:\> $single_proc

Handles  NPM(K)   PM(K)      WS(K)  VM(M)   CPU(s)  Id  ProcessName
-----  -----   -----      -----  -----   -----  --  -----
98        9       1340      7952    92     0.05  2756  notepad

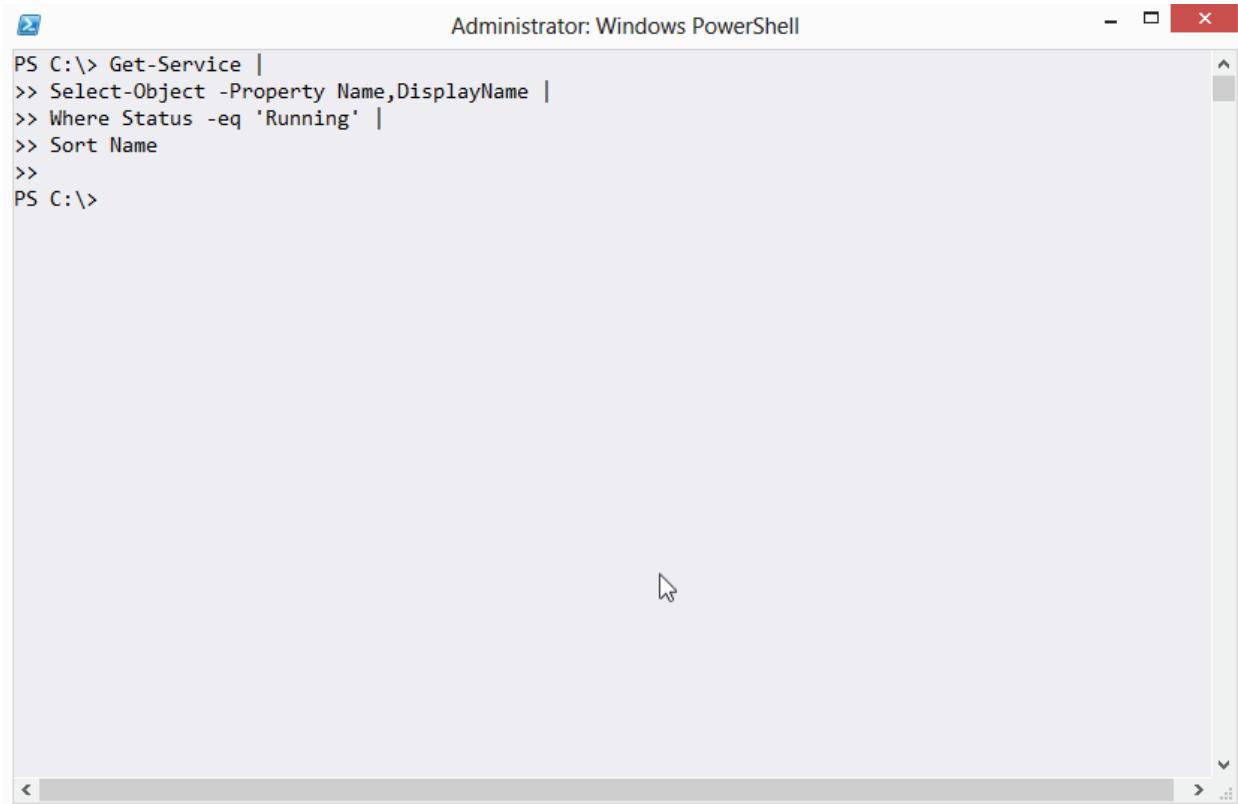
PS C:\> (get-process) -contains $single_proc
False
PS C:\> ■
```

In this example, I've started Notepad. I've put its process object into \$single_proc, and you can see that I verified it was there. But when I run Get-Process and check to see if its collection contained my Notepad, I got False. That's because the object in \$single_proc is out of date. Notepad is running, but it *now* looks different, so -contains can't find the match.

The -in and -contains operators are best with simple values, or with objects that don't have constantly -changing property values. But they're *not* wild card string matching operators. Use -like (or -notlike) for that.

You Can't Have What You Don't Have

Can you see what's wrong with this approach?



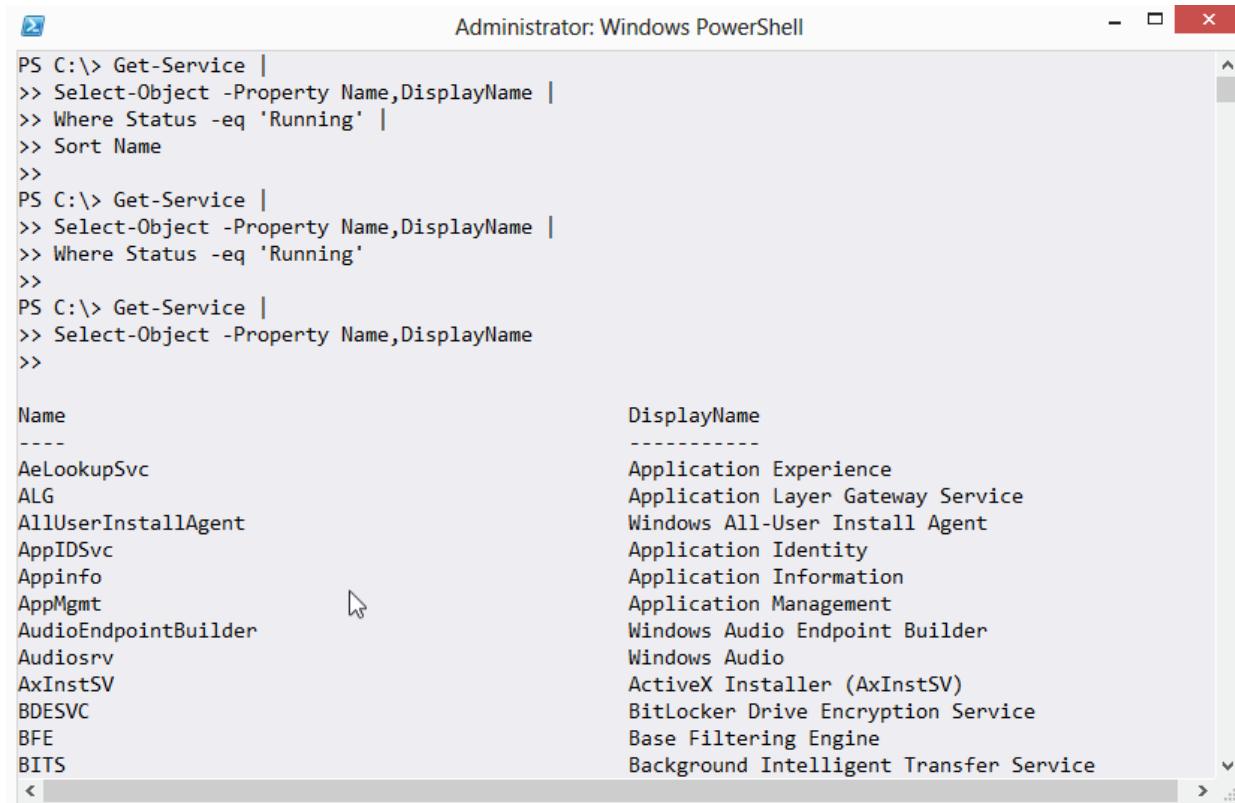
A screenshot of an Administrator Windows PowerShell window. The title bar reads "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> Get-Service |  
  >> Select-Object -Property Name,DisplayName |  
  >> Where Status -eq 'Running' |  
  >> Sort Name  
  >>  
PS C:\>
```

The command uses管道 (|) to chain several cmdlets: Get-Service, Select-Object, Where, and Sort. The final command PS C:\> is also shown.

I mean, I'm pretty sure I have some running services, which is what this was supposed to display.

If you don't see the answer right away - or frankly, even if you do - this is a good time to talk about how to troubleshoot long command lines. Start, as I always say, by *Backing off a step*. Delete the last command, and see if that does anything different.



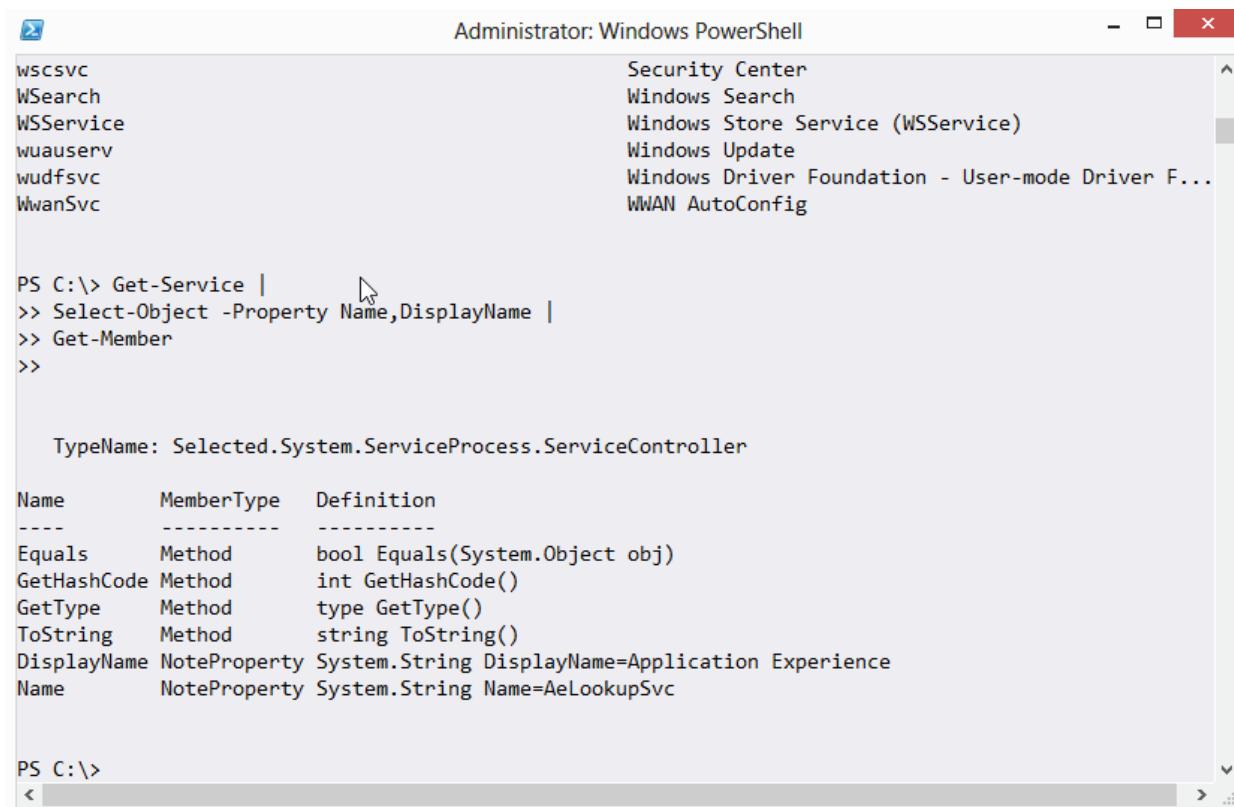
The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered was:

```
PS C:\> Get-Service |  
>> Select-Object -Property Name,DisplayName |  
>> Where Status -eq 'Running' |  
>> Sort Name  
>>  
PS C:\> Get-Service |  
>> Select-Object -Property Name,DisplayName |  
>> Where Status -eq 'Running'  
>>  
PS C:\> Get-Service |  
>> Select-Object -Property Name,DisplayName  
>>
```

The output displays a table of running services:

Name	DisplayName
AeLookupSvc	Application Experience
ALG	Application Layer Gateway Service
AllUserInstallAgent	Windows All-User Install Agent
AppIDSvc	Application Identity
Appinfo	Application Information
AppMgmt	Application Management
AudioEndpointBuilder	Windows Audio Endpoint Builder
Audiosrv	Windows Audio
AxInstSV	ActiveX Installer (AxInstSV)
BDESVC	BitLocker Drive Encryption Service
BFE	Base Filtering Engine
BITS	Background Intelligent Transfer Service

In this case, I removed the Sort-Object (Sort) command, and nothing different happened. So that wasn't causing the problem. Next, I removed the Where-Object (Where, using v3 short syntax) command, and ah-ha! I got output. So something broke with Where-Object. Let's take what *did* work and pipe it to Get-Member, to see what's in the pipeline after Select-Object runs.



Administrator: Windows PowerShell

```
PS C:\> Get-Service |  
>> Select-Object -Property Name,DisplayName |  
>> Get-Member  
>>  
  
TypeName: Selected.System.ServiceProcess.ServiceController  
  
Name      MemberType  Definition  
----      -----  
Equals    Method     bool Equals(System.Object obj)  
GetHashCode Method     int GetHashCode()  
GetType    Method     type GetType()  
ToString   Method     string ToString()  
DisplayName NoteProperty System.String DisplayName=Application Experience  
Name       NoteProperty System.String Name=AeLookupSvc  
  
PS C:\>
```

OK, I have an object that has a `DisplayName` property and a `Name` property.

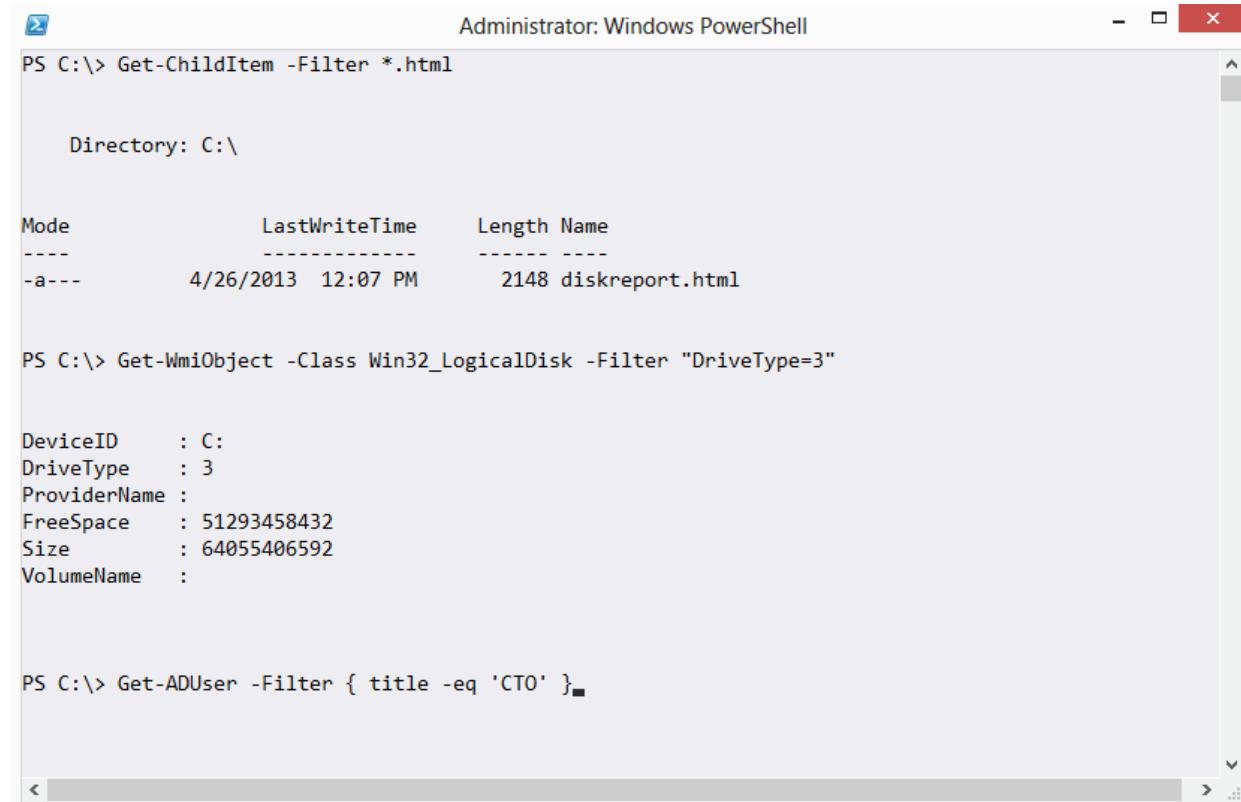
And my Where-Object command was checking the Status property. Do you see a Status property? No, you do not. My error is that I removed the Status property when I didn't include it in the property list of Select-Object. So Where-Object had nothing to work with, so it returned nothing.

(Yeah, it'd be cooler if it threw an error - "Hey, you said to filter on the Status property, and there ain't one!" - but that isn't how it works.)

Moral of the story: Pay attention to what's in the pipeline. You can't work with something you don't have, and you might have taken it away yourself. You won't always get a helpful error message, so sometimes you'll need to dig in and figure it out another way - such as backing off a step.

-Filter Values Diversity

Here's one of the toughest things to get used to in PowerShell:



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell" with the following content:

```
PS C:\> Get-ChildItem -Filter *.html
Directory: C:\

Mode                LastWriteTime     Length Name
----                -----          ---- 
-a---        4/26/2013 12:07 PM      2148 diskreport.html

PS C:\> Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3"

DeviceID      : C:
DriveType     : 3
ProviderName  :
FreeSpace     : 51293458432
Size          : 64055406592
VolumeName    :

PS C:\> Get-ADUser -Filter { title -eq 'CTO' }
```

Here you see three commands, each using a -Filter parameter. Every one of those filters is different.

With Get-ChildItem, -Filter accepts file system wildcards like *.

With Get-WmiObject, -Filter requires a string, and uses programming-style operators (like = for equality).

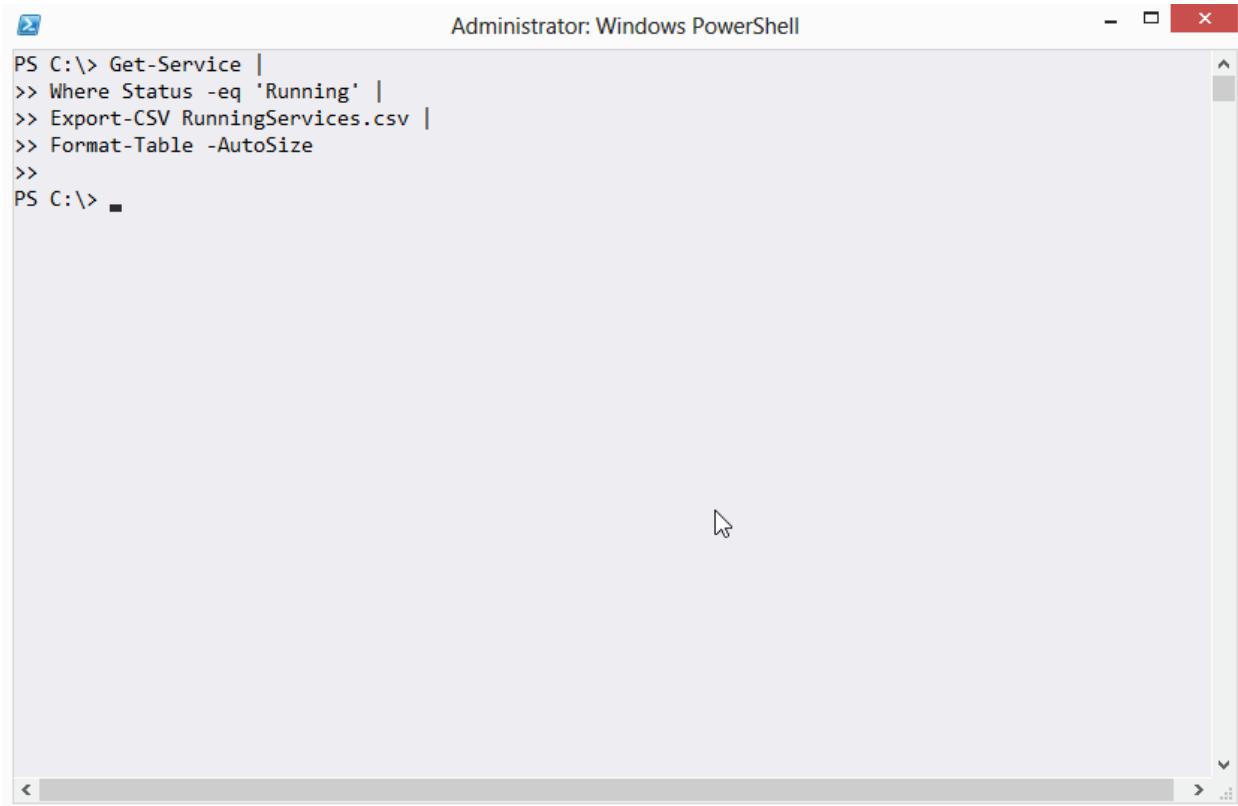
With Get-ADUser, -Filter wanted a script block, and accepted PowerShell-style comparison operators (like -eq for equality).

Here's how I think of it: When you use a -Filter parameter, PowerShell isn't processing the filtering. Instead, the filtration criteria is being handed down to the underlying technology, like the file system, or WMI, or Active Directory. *That* technology gets to decide what kind of filter criteria it will accept. PowerShell is just the middleman. So you have to carefully read the help, and maybe look for examples, to understand how the underlying technology needs you to specify its filter.

Yeah, it'd be nice if PowerShell just translated for you (that's actually what Get-ADUser does - the command translates that into an LDAP filter under the hood). But, usually, it doesn't.

Not Everything Produces Output

I see this one a lot in classes:



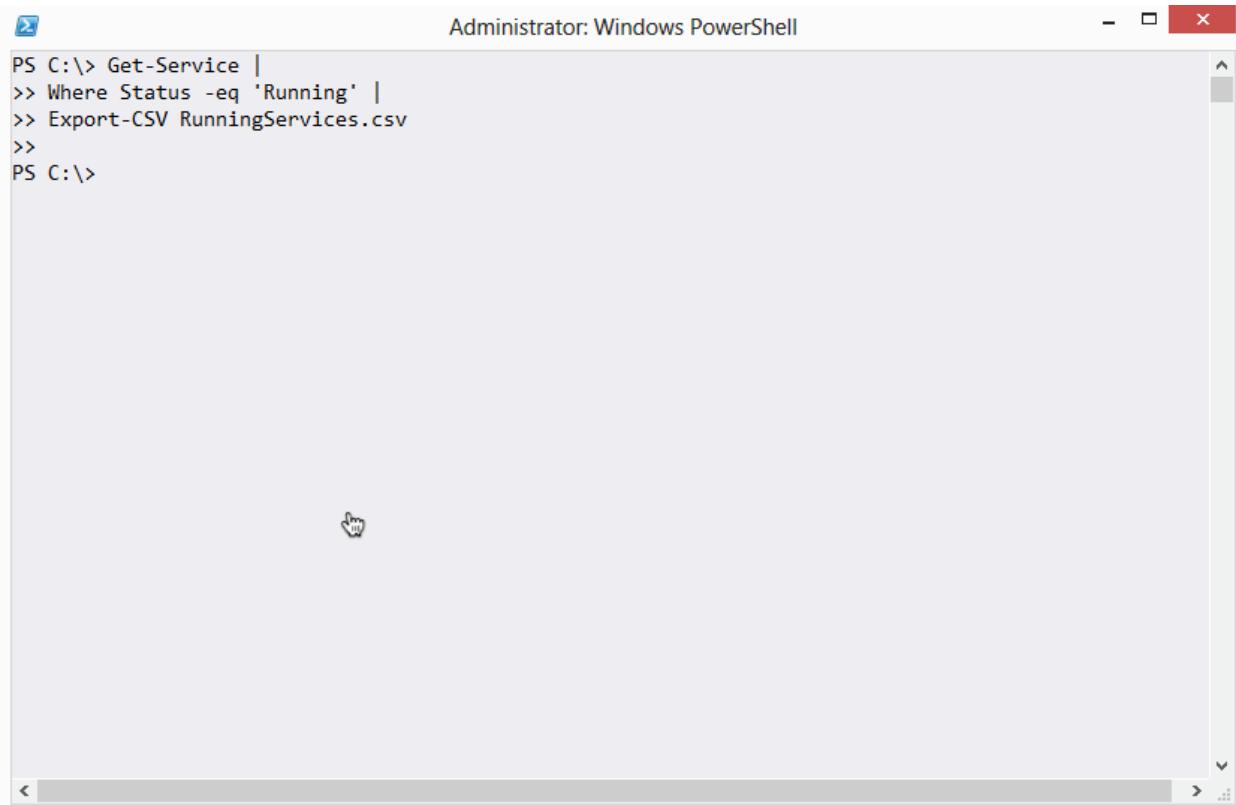
A screenshot of an Administrator Windows PowerShell window. The title bar reads "Administrator: Windows PowerShell". The command entered in the session is:

```
PS C:\> Get-Service |  
  >> Where Status -eq 'Running' |  
  >> Export-Csv RunningServices.csv |  
  >> Format-Table -AutoSize  
>>  
PS C:\>
```

The cursor is visible at the bottom center of the window.

If you expected anything on the screen in terms of output, you'd be disappointed. The trick here is to keep track of what each command produces as *output*, and right there is a possible point of confusion.

In PowerShell's world, *output* is what would show up on the screen if you ran the command and didn't pipe it to anything else. Yes, `Export-Csv` does do something - it creates a file on disk - but in PowerShell's world that file isn't *output*. What `Export-Csv` does *not* do is produce any output - that is, something which would show up on the screen. For example:



A screenshot of an Administrator Windows PowerShell window. The title bar reads "Administrator: Windows PowerShell". The command entered in the console is:

```
PS C:\> Get-Service |  
  >> Where Status -eq 'Running' |  
  >> Export-Csv RunningServices.csv  
>>  
PS C:\>
```

The window shows a single line of the command being typed, with the cursor positioned at the end of the final pipe operator. The rest of the window is blank, indicating that no output has been generated by the command.

See? Nothing. Since there's nothing on the screen, there's nothing *in the pipeline*. You can't pipe Export-Csv to another command, *because there's nothing to pipe*.

Some commands will include a -PassThru parameter. When they have one, and when you use it, they'll do whatever they normally do but *also* pass their input objects through to the pipeline, so that you can then pipe them on to something else. Export-Csv isn't one of those commands, though - it never produces output, so it will never make sense to pipe it to something else.



SAPIEN Technologies, Inc. 2014 PRODUCTS



PrimalScript 2014

The industry-standard universal scripting IDE. With support for over 50 languages and file types. No administrator can live without this.

Price: \$ 389.00 **SKU# PSR14-SR**



PowerShell Studio 2014

The PowerShell Toolmaking IDE. Transform your administrative scripts into real Windows applications. Create PowerShell scripts, GUIs, modules, executables, and MSI installers.

Price: \$ 389.00 **SKU# SPS14-SR**



VersionRecall 2014

The simplest way to manage multiple versions of files on your computer.

Price: \$ 179.00 **SKU# VRK14-SR**



ChangeVue 2014

Version control made easy. Keep your files safe and track your changes in this versatile source control tool for small teams.

Price: \$ 179.00 **SKU# VUE14-SR**



PrimalXML 2014

A dedicated editor for this universal file format provides all the standard tools you expect and more. Validate your XML files against your schema and easily format generated XML into readable text.

Price: \$ 89.00 **SKU# PXL14-SR**



PrimalSQL 2014

Connect to any database and create and execute SQL queries from one easy tool. A graphical query builder allows even a novice to create complex queries within minutes.

Price: \$ 89.00 **SKU# PQL14-SR**



The SAPIEN Software Suite 2014 (S3):

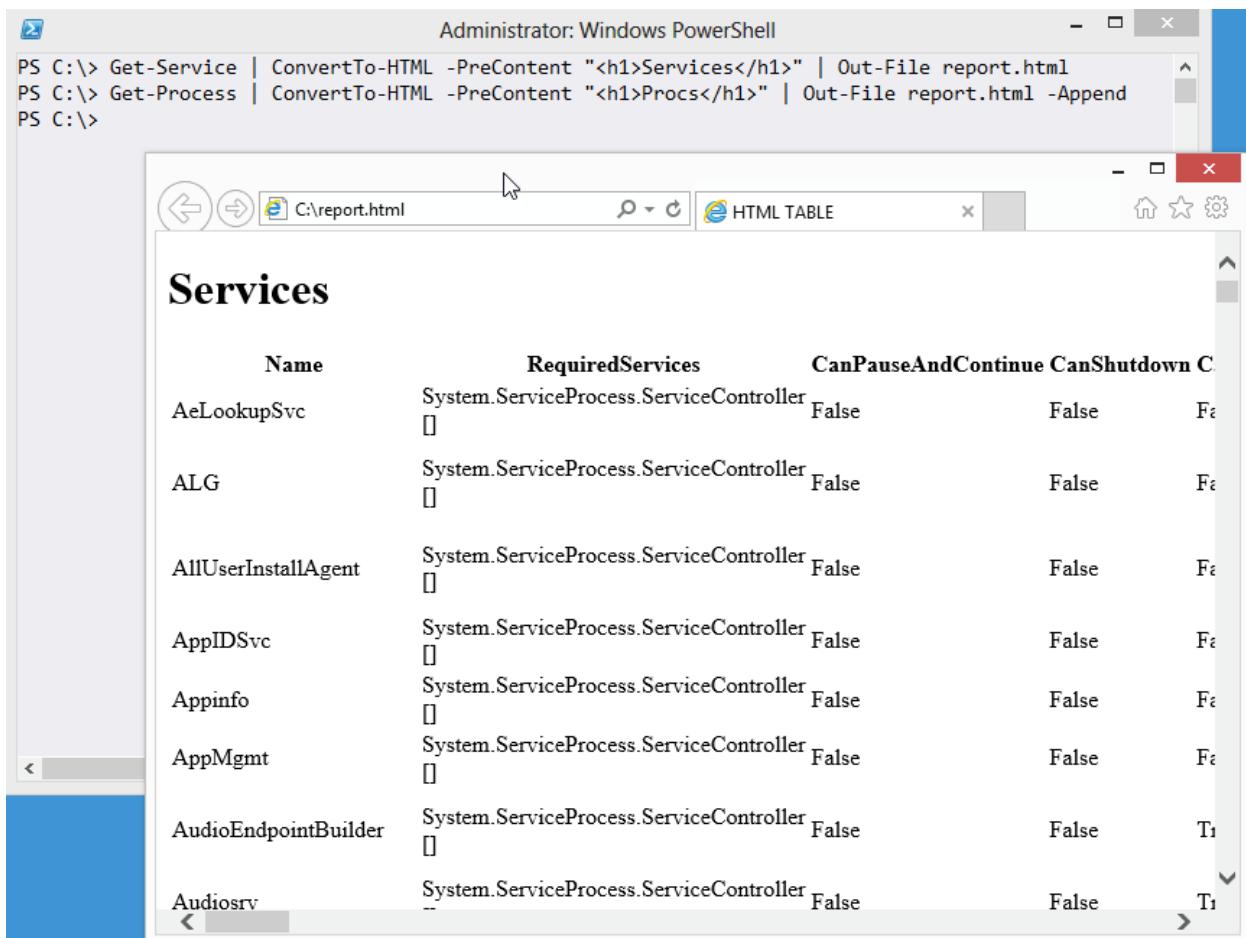
Offers current versions of all of our desktop software tools for one low price. The following products are included:

- PrimalScript 2014
- PowerShell Studio 2014
- PrimalXML 2014
- PrimalSQL 2014
- ChangeVue 2014
- VersionRecall 2014



One HTML Page at a Time, Please

This drives me batty:



Administrator: Windows PowerShell

```
PS C:\> Get-Service | ConvertTo-HTML -PreContent "<h1>Services</h1>" | Out-File report.html
PS C:\> Get-Process | ConvertTo-HTML -PreContent "<h1>Procs</h1>" | Out-File report.html -Append
PS C:\>
```

C:\report.html

HTML TABLE

Services

Name	RequiredServices	CanPauseAndContinue	CanShutdown	CanStop
AeLookupSvc	System.ServiceProcess.ServiceController	False	False	False
ALG	System.ServiceProcess.ServiceController	False	False	False
AllUserInstallAgent	System.ServiceProcess.ServiceController	False	False	False
AppIDSvc	System.ServiceProcess.ServiceController	False	False	False
Appinfo	System.ServiceProcess.ServiceController	False	False	False
AppMgmt	System.ServiceProcess.ServiceController	False	False	False
AudioEndpointBuilder	System.ServiceProcess.ServiceController	False	False	True
Audiosrv	System.ServiceProcess.ServiceController	False	False	True

What's happening is that someone ran two command, piping the output of each to ConvertTo-HTML, and essentially sticking both HTML pages into a single file. What drives me really nuts is that Internet Explorer is okay with that nonsense.

HTML files are allowed to start with one top-level <HTML> tag, but if you check out that file you'll see that it contains two. Here's the middle bit:

```
Administrator: Windows PowerShell
PS C:\> Get-Service | ConvertTo-HTML -PreContent "<h1>Services</h1>" | Out-File report.html
PS C:\> Get-Process | ConvertTo-HTML -PreContent "<h1>Procs</h1>" | Out-File report.html -Append
PS C:\> notepad .\report.html
PS C:\>
```

report - Notepad

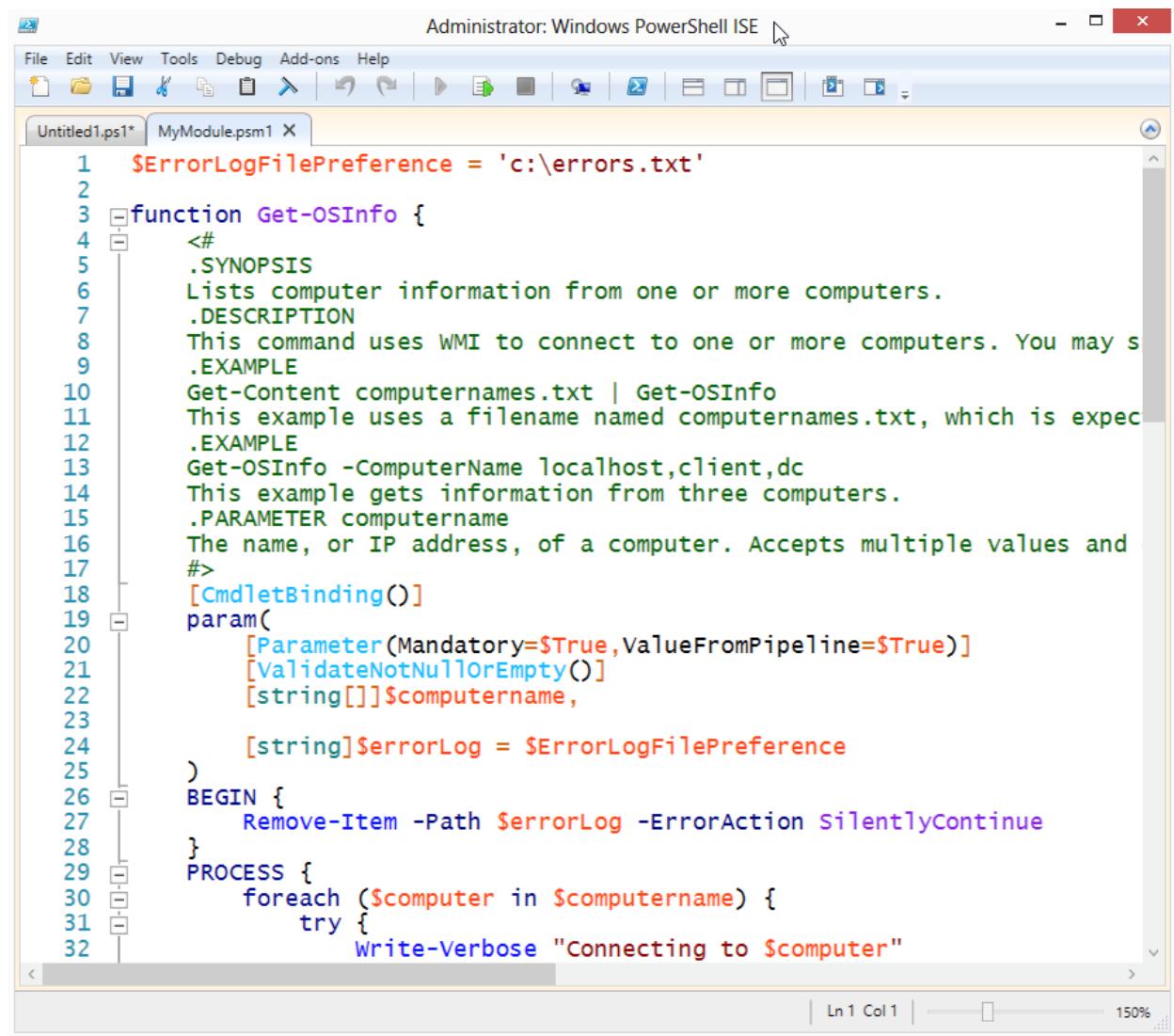
```
<tr><td>WwanSvc</td><td>System.ServiceProcess.ServiceController<br/>[]</td><td>False</td><td>False</td><td>False</td><td>WWAN<br/>AutoConfig</td><td>System.ServiceProcess.ServiceController<br/>[]</td><td>.</td><td>WwanSvc</td><td>System.ServiceProcess.ServiceController<br/>[]</td><td>SafeServiceHandle</td><td>Stopped</td><td>Win32ShareProvider</td></tr>
</table>
</body></html>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>HTML TABLE</title>
</head><body>
<h1>Procs</h1>
<table>
```

I've highlighted the lines that end one HTML page and start the next one. This is technically a malformed HTML file. It becomes tough to use this with some Web browsers (Firefox 20 is choking it down, but my current Webkit browsers aren't), tough to parse if you ever need to manipulate it programmatically, and... well, it's just a bad thing. It's like incest or something. Gross.

If you need to combine multiple elements into a single HTML file, you use the `-Fragment` switch of `ConvertTo-HTML`. That produces just a portion of the HTML, and you can produce several such portions and then combine them into a single, complete page. Ahhh, nice. That whole process is covered in *Creating HTML Reports in PowerShell*, another free ebook that came with this one.

[Bloody] {Awful} (Punctuation)

This isn't so much a "gotcha" as it is just plain confusing. PowerShell's nuts with the punctuation.



```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Untitled1.ps1* MyModule.psm1 X
1 $ErrorLogFilePreference = 'c:\errors.txt'
2
3 function Get-OSInfo {
4     <#
5     .SYNOPSIS
6     Lists computer information from one or more computers.
7     .DESCRIPTION
8     This command uses WMI to connect to one or more computers. You may s
9     .EXAMPLE
10    Get-Content computernames.txt | Get-OSInfo
11    This example uses a filename named computernames.txt, which is expec
12    .EXAMPLE
13    Get-OSInfo -ComputerName localhost,client,dc
14    This example gets information from three computers.
15    .PARAMETER computername
16    The name, or IP address, of a computer. Accepts multiple values and
17    #>
18    [CmdletBinding()]
19    param(
20        [Parameter(Mandatory=$True,ValueFromPipeline=$True)]
21        [ValidateNotNullOrEmpty()]
22        [string[]]$computername,
23
24        [string]$errorLog = $ErrorLogFilePreference
25    )
26    BEGIN {
27        Remove-Item -Path $errorLog -ErrorAction SilentlyContinue
28    }
29    PROCESS {
30        foreach ($computer in $computername) {
31            try {
32                Write-Verbose "Connecting to $computer"
```

(Parentheses) are used to enclose expressions, such as the `ForEach()` construct's expression, and in certain cases to contain declarative syntax. You see that in the `Param()` block, and in the `[Parameter()]` attribute.

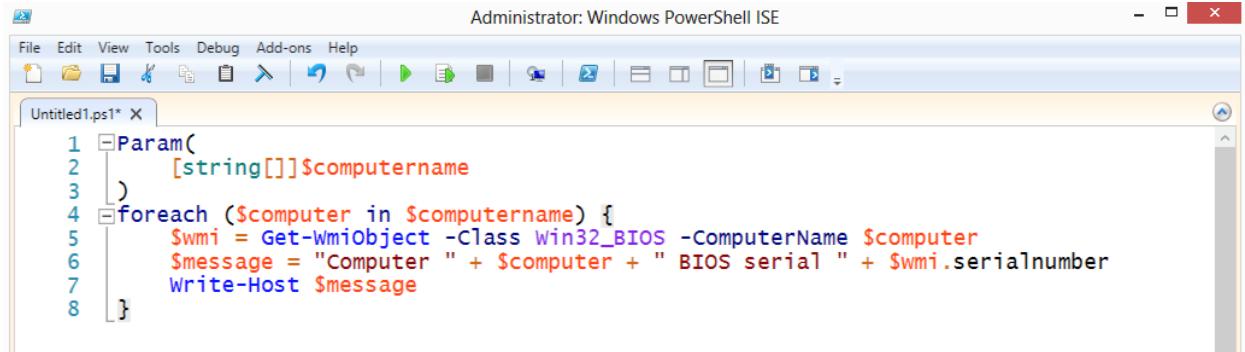
[Square brackets] are used around some attributes, like `[CmdletBinding()]`, and around data types like `[string]`, and to indicate arrays - as in `[string[]]`. They pop up a few other places, too.

{Curly brackets } nearly always contain executable code, as in the `Try{ }` block, the `BEGIN{ }` block, and the function itself. It's also used to express hash table literals (like `@{}`).

If your keyboard had a few dozen more buttons, PowerShell probably wouldn't have had to have all these overlapping uses of punctuation. But it does. At this point, they're pretty much just part of the shell's "cost of entry," and you'll have to get used to them.

Don't Concatenate Strings

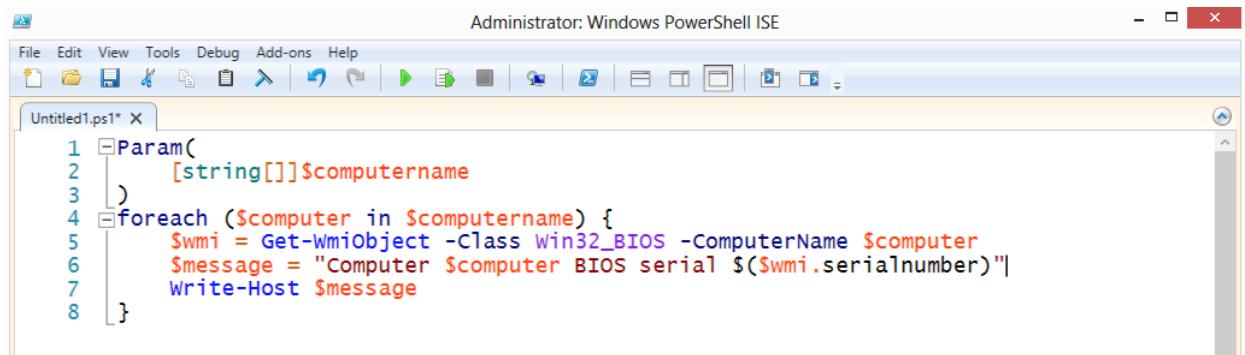
I really dislike string concatenation. It's like *forcing* someone to cuddle with someone they don't even know. Rude.



A screenshot of the Windows PowerShell ISE interface. The title bar says "Administrator: Windows PowerShell ISE". The code editor window shows a file named "Untitled1.ps1" with the following content:

```
1 Param(
2     [string[]]$computername
3 )
4 foreach ($computer in $computername) {
5     $wmi = Get-WmiObject -Class Win32_BIOS -ComputerName $computer
6     $message = "Computer " + $computer + " BIOS serial " + $wmi.serialnumber
7     Write-Host $message
8 }
```

And completely unnecessary, when you use double quotes.



A screenshot of the Windows PowerShell ISE interface. The title bar says "Administrator: Windows PowerShell ISE". The code editor window shows a file named "Untitled1.ps1" with the following content:

```
1 Param(
2     [string[]]$computername
3 )
4 foreach ($computer in $computername) {
5     $wmi = Get-WmiObject -Class Win32_BIOS -ComputerName $computer
6     $message = "Computer $computer BIOS serial $($wmi.serialnumber)"|
7     Write-Host $message
8 }
```

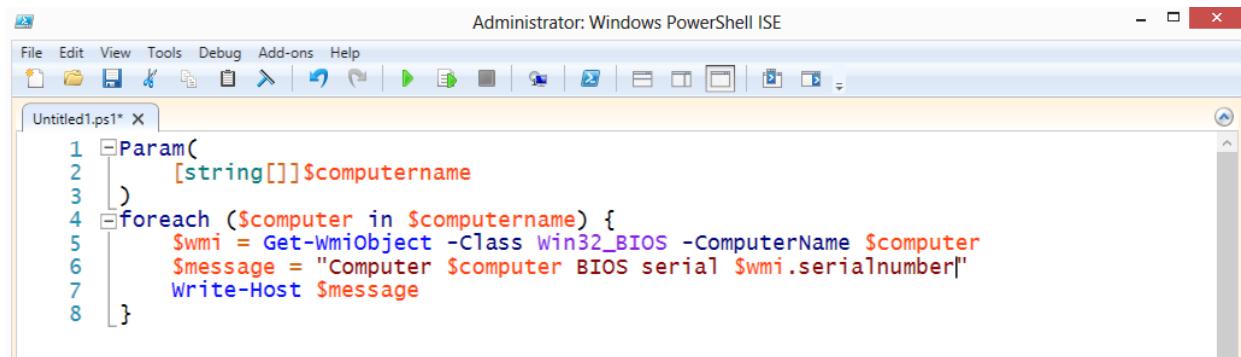
Same end effect. In double quotes, PowerShell will look for the \$ character. When it finds it:

If the next character is a { then PowerShell will take everything to the matching } as a variable name, and replace the whole thing with that variable's contents. For example, putting \${my variable} inside double quotes will replace that with the contents of \${my variable}.

If the next character is a (then PowerShell will take everything to the matching) and execute it as code. So, I executed \$wmi.serialnumber to access the serialnumber property of whatever object was in the \$wmi variable.

Otherwise, PowerShell will take every character that is legal for a variable name, up until the first illegal variable name character, and replace it with that variable. That's how \$computer works in my example. The space after r isn't legal for a variable name, so PowerShell knows the variable name stops at r.

There's a sub-gotcha here:



A screenshot of the Windows PowerShell Integrated Scripting Environment (ISE). The title bar says "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar has icons for file operations like Open, Save, and Run. The main window shows an untitled script named "Untitled1.ps1". The code is as follows:

```
1 Param(
2     [string[]]$computername
3 )
4 foreach ($computer in $computername) {
5     $wmi = Get-WmiObject -Class Win32_BIOS -ComputerName $computer
6     $message = "Computer $computer BIOS serial $wmi.serialnumber"
7     Write-Host $message
8 }
```

This won't work as expected. In most cases, `$wmi` will be replaced by an object type name, and `.serialnumber` will still be in there. That's because `.` isn't a legal variable name character, so PowerShell stops looking at the variable with the letter `i`. It replaces `$wmi` with its contents. You see, in the previous example, I'd put `$(($wmi.serialnumber))`, which is a *subexpression*, and which works. The parentheses make their contents execute as code.

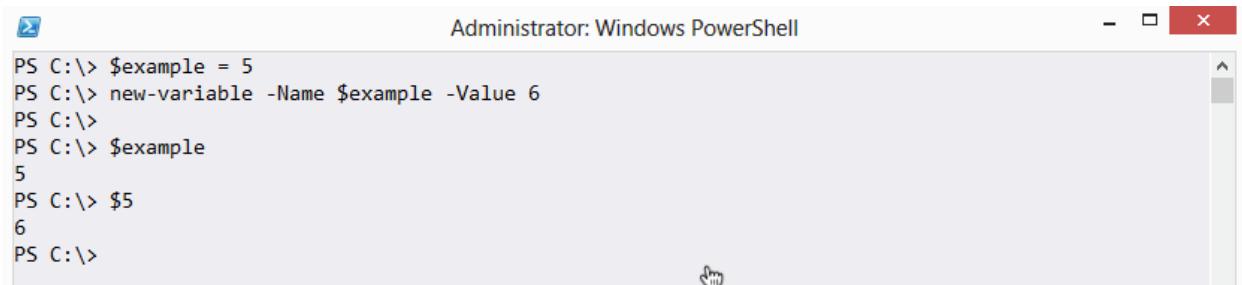
\$ isn't Part of the Variable Name

Big gotcha.



```
Administrator: Windows PowerShell
PS C:\> $example = 5
PS C:\> new-variable -Name $example -Value 6
PS C:\>
```

Can you predict what happened?

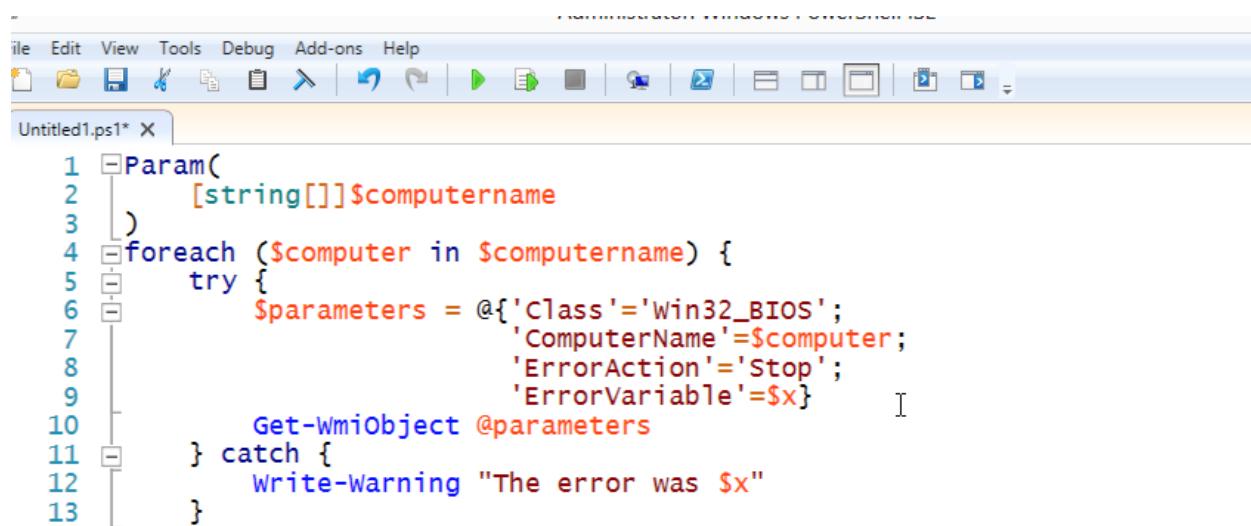


```
Administrator: Windows PowerShell
PS C:\> $example = 5
PS C:\> new-variable -Name $example -Value 6
PS C:\>
PS C:\> $example
5
PS C:\> $5
6
PS C:\>
```

You see, the \$ is not part of the variable's name. If you have a variable named **example**, that's like having a box with "example" written on the side. Referring to **example** means you're talking *about the box itself*. Referring to **\$example** means you're messing with the *contents of the box*.

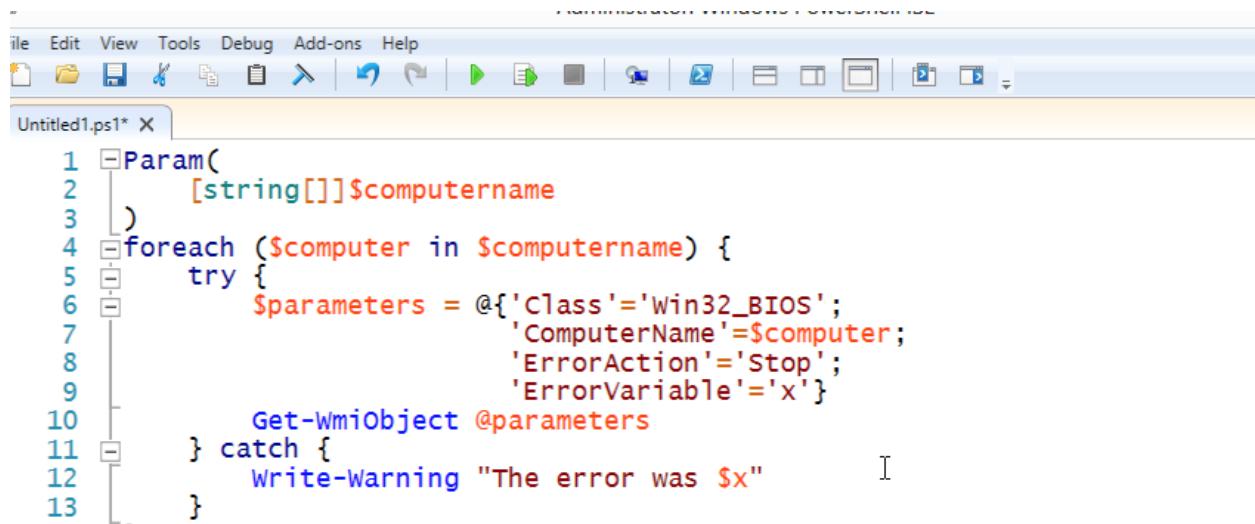
So in my example, I used **\$example=5** to put 5 *into the box*. I then created a new variable. The new variable's name was **\$example** - that isn't *naming it* "example," it's *naming it the contents of the "example" box*, which is 5. So I create a variable named 5, that contains 6, which you can see by referring to **\$5**.

Tricky, right? Comes up all the time:



```
Administrator: Windows PowerShell
File Edit View Tools Debug Add-ons Help
Untitled1.ps1* X
1 Param(
2     [string[]]$computername
3 )
4 foreach ($computer in $computername) {
5     try {
6         $parameters = @{
7             'Class'='Win32_BIOS';
8             'ComputerName'=$computer;
9             'ErrorAction'='Stop';
10            'ErrorVariable'=$x}
11         Get-WmiObject @parameters
12     } catch {
13         Write-Warning "The error was $x"
14     }
15 }
```

In that example, I used the **-ErrorVariable** parameter to specify a variable in which I would store any error that would occur. Problem is, I used **\$x**. I should have used **x** by itself:



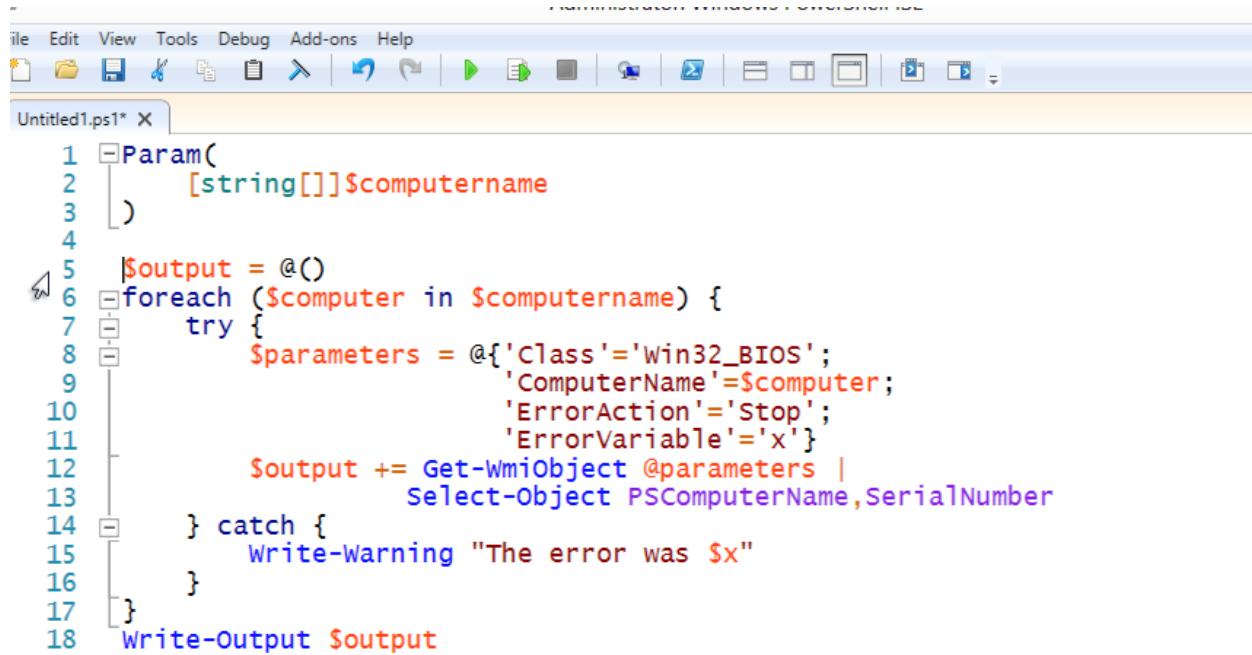
A screenshot of the Notepad++ text editor. The title bar says "Untitled1.ps1*". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, Help. The toolbar has icons for file operations like Open, Save, Find, Copy, Paste, etc. The code in the editor is:

```
1 Param(
2     [string[]]$computername
3 )
4 foreach ($computer in $computername) {
5     try {
6         $parameters = @{
7             'Class'='Win32_BIOS';
8             'ComputerName'=$computer;
9             'ErrorAction'='Stop';
10            'ErrorVariable'='x'
11        }
12        Get-WmiObject @parameters
13    } catch {
14        Write-Warning "The error was $x"
15    }
16 }
```

That will store any error in a variable named `x`, which I can later access by using `$x` to get its contents - meaning, whatever error was stored in there.

Use the Pipeline, not an Array

A very common mistake made by traditional programmers who come to PowerShell - which is *not* a programming language:

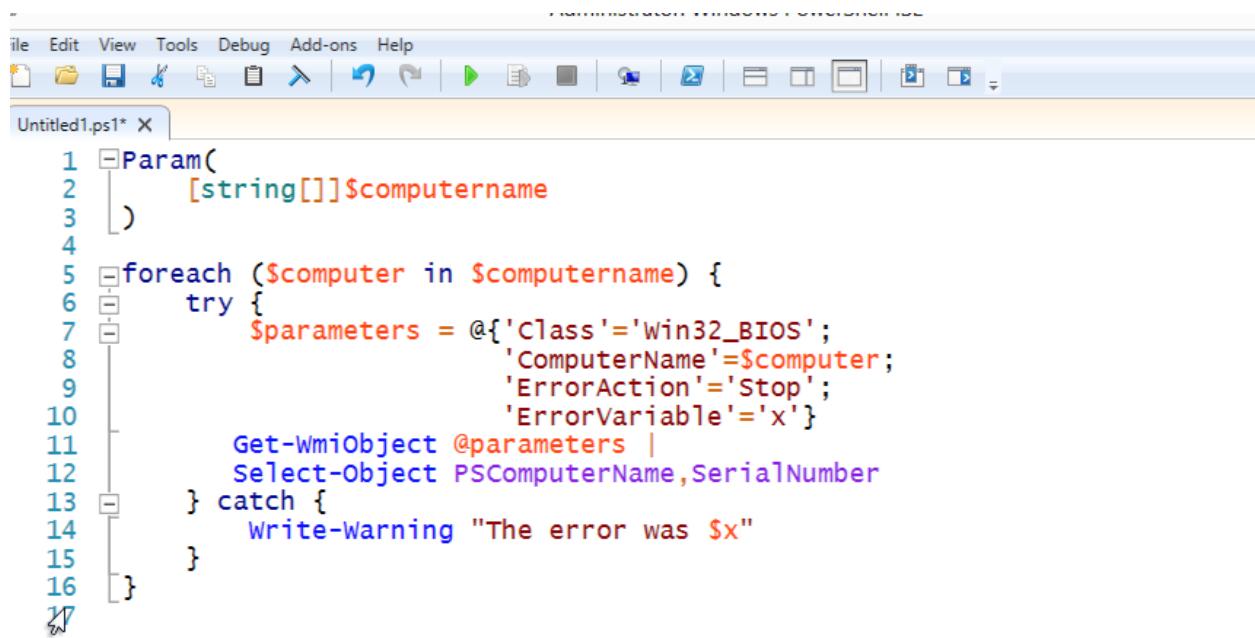


The screenshot shows a Notepad window titled "Untitled1.ps1". The code is as follows:

```
1 Param(
2     [string[]]$computername
3 )
4
5 $output = @()
6 foreach ($computer in $computername) {
7     try {
8         $parameters = @{
9             'Class'='Win32_BIOS';
10            'ComputerName'=$computer;
11            'ErrorAction'='Stop';
12            'ErrorVariable'='x'}
13         $output += Get-WmiObject @parameters |
14             Select-Object PSComputerName,SerialNumber
15     } catch {
16         Write-Warning "The error was $x"
17     }
18 }
19 Write-Output $output
```

This person has created an empty array in **\$output**, and as they run through their computer list and query WMI, they're adding new output objects to the array. Finally, at the end, they output the array to the pipeline.

Poor practice. You see, this forces PowerShell to wait while this *entire* command completes. Any subsequent commands in the pipeline will sit there twiddling their thumbs. A better approach? Use the pipeline. Its whole *purpose* is to accumulate output for you - there's no need to accumulate it yourself in an array.



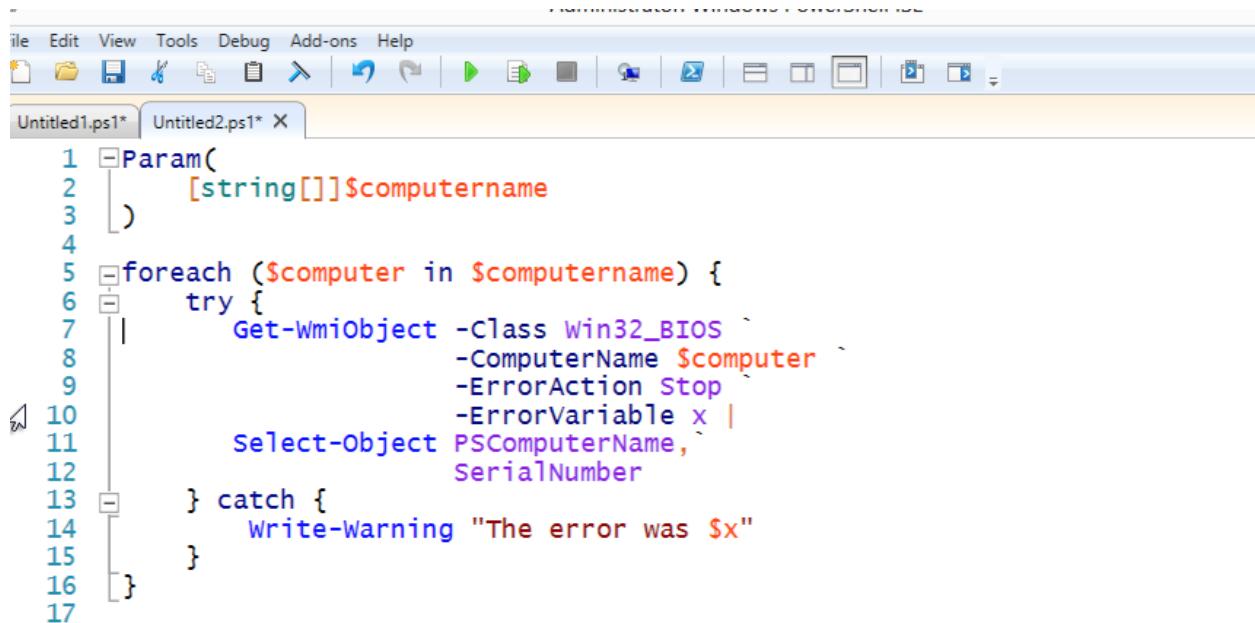
A screenshot of a Windows Notepad window titled "Untitled1.ps1". The window contains the following PowerShell script:

```
1 Param(
2     [string[]]$computername
3 )
4
5 foreach ($computer in $computername) {
6     try {
7         $parameters = @{
8             'Class'='Win32_BIOS';
9             'ComputerName'=$computer;
10            'ErrorAction'='Stop';
11            'ErrorVariable'='x'}
12
13         Get-WmiObject @parameters |
14         Select-Object PSComputerName,SerialNumber
15     } catch {
16         Write-Warning "The error was $x"
17     }
18 }
```

Now, subsequent commands will receive output *as its being created*, letting several commands run more or less simultaneously in the pipeline.

Backtick, Grave Accent, Escape

You'll see folks do this a lot:



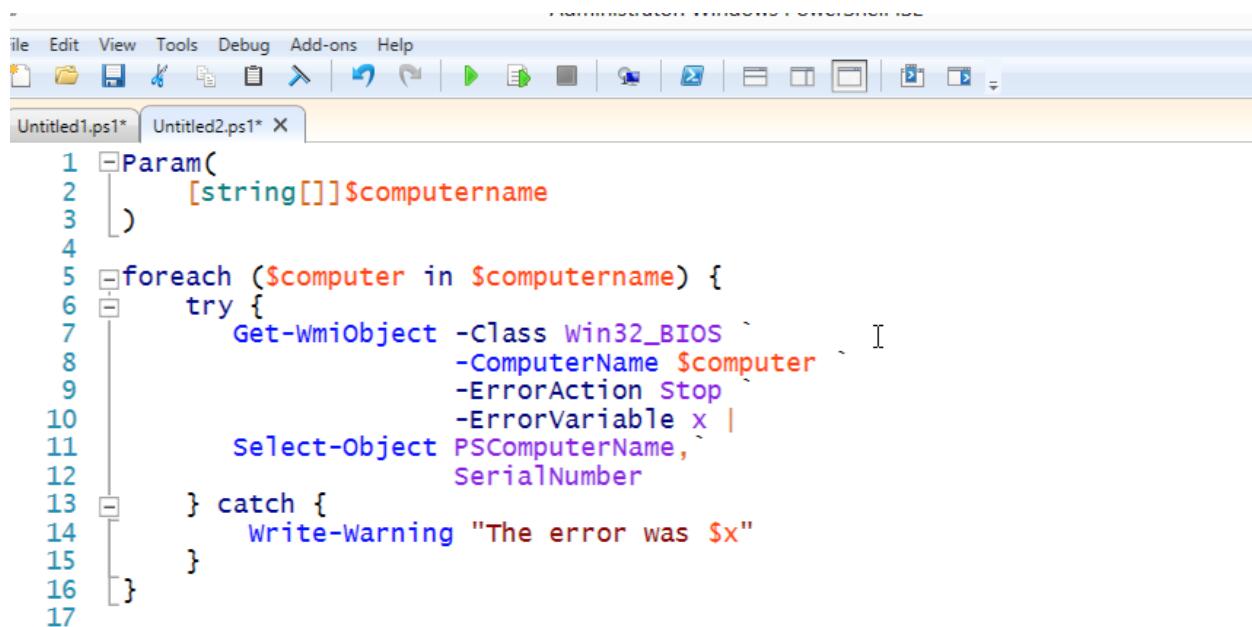
A screenshot of the Windows PowerShell ISE interface. The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar contains various icons for file operations. Two tabs are open: Untitled1.ps1* and Untitled2.ps1*. The code in Untitled1.ps1* is as follows:

```
1 Param(
2     [string[]]$computername
3 )
4
5 foreach ($computer in $computername) {
6     try {
7         Get-WmiObject -Class Win32_BIOS
8             -ComputerName $computer
9             -ErrorAction Stop
10            -ErrorVariable x |
11                Select-Object PSCoMPuterName,
12                    SerialNumber
13    } catch {
14        Write-Warning "The error was $x"
15    }
16
17 }
```

That isn't a dead pixel on your monitor or a stray piece of toner on the page, it's the *grave accent mark* or *backtick*. ` is PowerShell's escape character. In this example, it's “escaping” the invisible carriage return at the end of the line, removing its special purpose as a logical line-end, and simply making it a literal carriage return.

I don't like the backtick used this way.

First, it's hard to see. Second, if you get any extra whitespace after it, it'll no longer escape the carriage return, and your script will break. The ISE even figures this out:



A screenshot of the Windows PowerShell ISE interface. The title bar says "Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar has icons for file operations like Open, Save, and Run. There are two tabs at the top: "Untitled1.ps1*" and "Untitled2.ps1* X". The main code editor area contains the following PowerShell script:

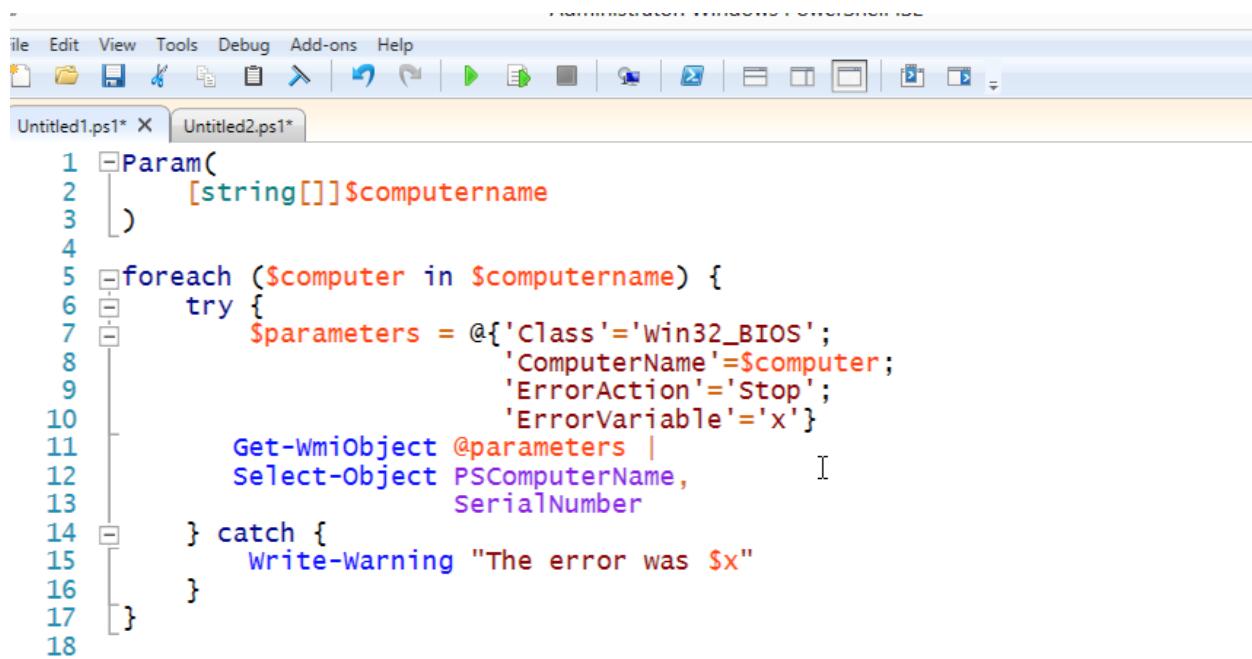
```
1 Param(
2     [string[]]$computername
3 )
4
5 foreach ($computer in $computername) {
6     try {
7         Get-WmiObject -Class Win32_BIOS
8             -ComputerName $computer
9             -ErrorAction Stop
10            -ErrorVariable x |
11                Select-Object PSCoputerName,
12                    SerialNumber
13    } catch {
14        Write-Warning "The error was $x"
15    }
16 }
17
```

Carefully compare the `-ComputerName` parameter - in this second example, it's the wrong color for a parameter name, because I added a space after the backtick on the preceding line. IMPOSSIBLE to track these down.

And the backtick is unnecessary as a line continuation character. Let me explain why:

PowerShell already allows you to hit Enter in certain situations. You just have to learn what those situations are, and learn to take advantage of them. I totally understand the desire to have neatly-formatted code - I preach about that all the time, myself - but you don't have to rely on a little three-pixel character to get nicely formatted code.

You just have to be clever.



A screenshot of a Windows Notepad window titled "Untitled2.ps1*". The window contains the following PowerShell script:

```
1 Param(
2     [string[]]$computername
3 )
4
5 foreach ($computer in $computername) {
6     try {
7         $parameters = @{
8             'Class'='Win32_BIOS';
9             'ComputerName'=$computer;
10            'ErrorAction'='Stop';
11            'ErrorVariable'='x'}
12
13         Get-WmiObject @parameters |
14         Select-Object PSCoMPuterName,
15                                         SerialNumber
16     } catch {
17         Write-Warning "The error was $x"
18     }
}
```

To begin, I've put my `Get-WmiObject` commands in a *hash table*, so I can format them all nice and pretty. Each line ends on a semicolon, and PowerShell lets me line-break after each semicolon. Even if I get an extra space or tab after the semicolon, it'll work fine. I then *splat* those parameters to the `Get-WmiObject` command.

After `Get-WmiObject`, I have a pipe character - and you can legally line-break after that, too.

You'll notice on `Select-Object` that breaking after a comma as well.

So I end up with formatting that looks at least as good, if not *better*, because it doesn't have that little ` floating all over the place.

Live Training by MVP Instructors in Phoenix or Online!



DON JONES

**DJPS710 - Don Jones' Exclusive
Accelerated PowerShell v3 Masterclass
for Administrators**



JASON HELMICK

**PS305 - PowerShell v3 Training
for Administrators**

IIS300 - Microsoft IIS

Attend class online from anywhere!
Call 602-266-8500 for a live demo.

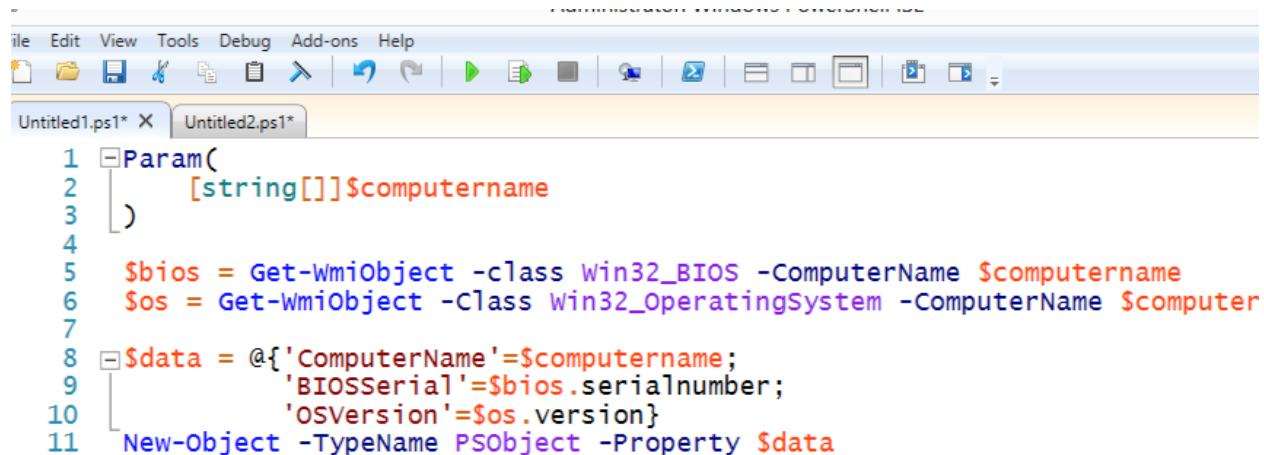


interface
www.InterfaceTT.com

WINDOWS SERVER 2012 • WINDOWS 7/8 • MICROSOFT SYSTEM CENTER • POWERSHELL
SHAREPOINT • EXCHANGE SERVER • SQL SERVER 2012 • CISCO • WEB DEVELOPMENT
ITIL • PMP • COBIT • CITRIX • COMPTIA • VMWARE • RED HAT LINUX

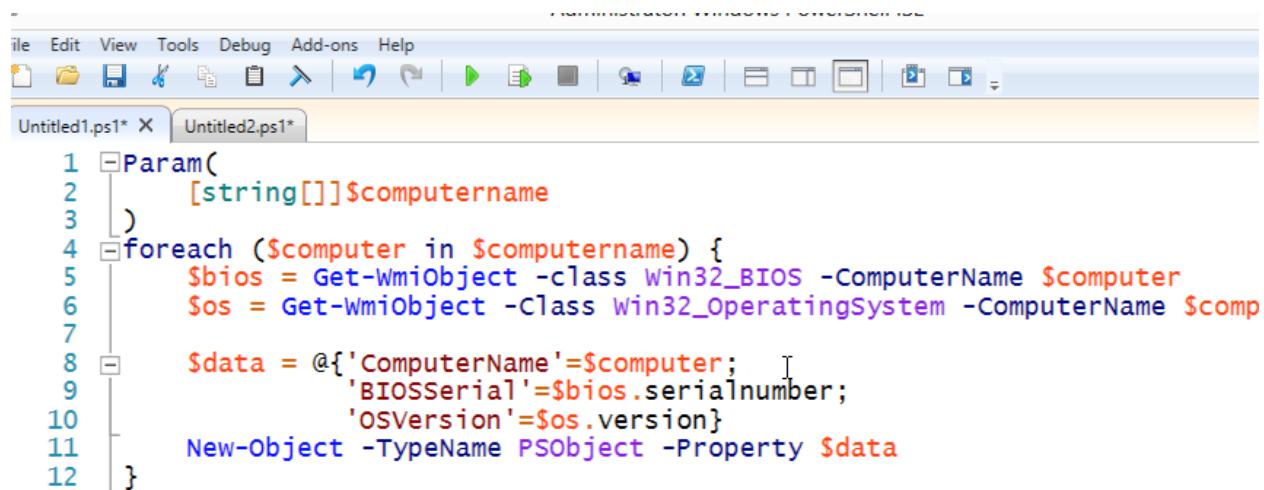
A Crowd isn't an Individual

A very common newcomer mistake:



```
1 Param(
2     [string[]]$computername
3 )
4
5 $bios = Get-WmiObject -class Win32_BIOS -ComputerName $computername
6 $os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName $computer
7
8 $data = @{'ComputerName'=$computername;
9           'BIOSSerial'=$bios.serialnumber;
10          'OSVersion'=$os.version}
11 New-Object -TypeName PSObject -Property $data
```

Here, the person is treating everything like it contains only one value. But \$computername might contain multiple computer names (that's what [string[]] means), meaning \$bios and \$os will contain multiple items too. You'll often have to enumerate those to get this working right:



```
1 Param(
2     [string[]]$computername
3 )
4 foreach ($computer in $computername) {
5     $bios = Get-WmiObject -class Win32_BIOS -ComputerName $computer
6     $os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName $comp
7
8     $data = @{'ComputerName'=$computer;
9               'BIOSSerial'=$bios.serialnumber;
10              'OSVersion'=$os.version}
11     New-Object -TypeName PSObject -Property $data
12 }
```

Folks will run into this even in simple situations. For example:

```
Administrator: Windows PowerShell
PS C:\> $procs = Get-Process
PS C:\> $message = "Process name $procs.name"
PS C:\> $message
Process name System.Diagnostics.Process (conhost) System.Diagnostics.Process (csrss) System.Diagnostics.Process (csrss) System.Diagnostics.Process (dwm) System.Diagnostics.Process (explorer) System.Diagnostics.Process (Idle) System.Diagnostics.Process (lsass) System.Diagnostics.Process (MsMpEng) System.Diagnostics.Process (powershell) System.Diagnostics.Process (powershell_ise) System.Diagnostics.Process (SearchIndexer) System.Diagnostics.Process (services) System.Diagnostics.Process (smss) System.Diagnostics.Process (spoolsv) System.Diagnostics.Process (svchost) System.Diagnostics.Process (TabTip) System.Diagnostics.Process (TabTip32) System.Diagnostics.Process (taskhost) System.Diagnostics.Process (taskhostex) System.Diagnostics.Process (vmtoolsd) System.Diagnostics.Process (vmtoolsd) System.Diagnostics.Process (wininit) System.Diagnostics.Process (winlogon).name
PS C:\>
```

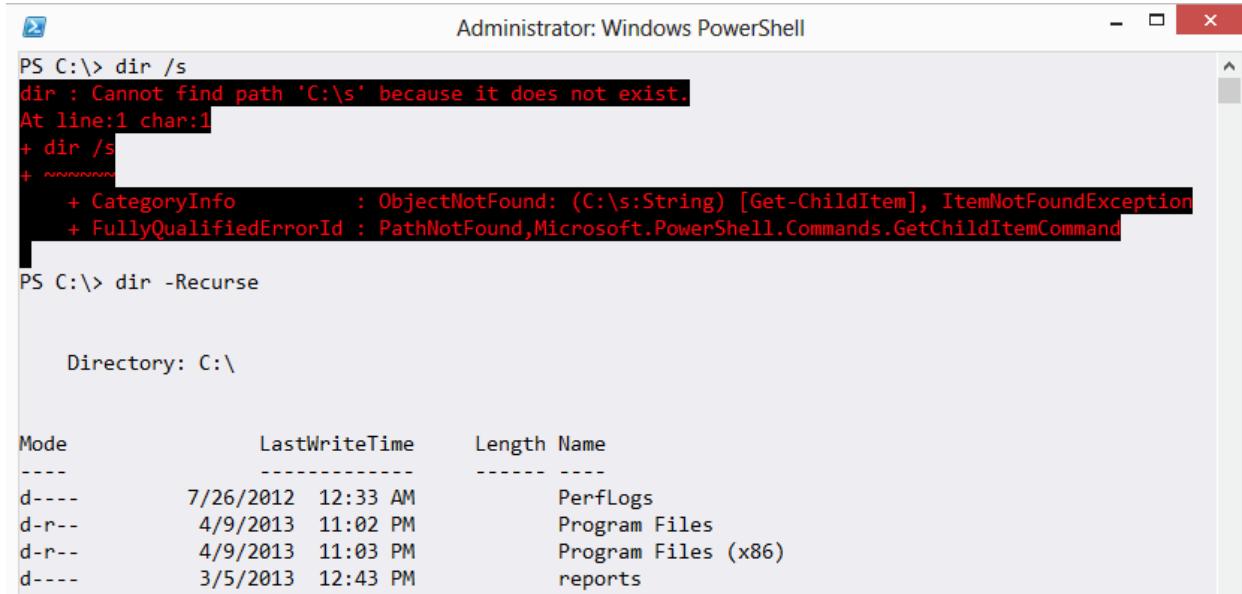
PowerShell v2 won't react so nicely; in v3, the variable inside double quotes is `$procs`, and since that variable contains multiple objects, PowerShell implicitly enumerates them and looks for a `Name` property. You'll notice "`.name`" from the original string appended to the end - PowerShell didn't do anything with that.

You'd probably want to enumerate these:

```
Administrator: Windows PowerShell
PS C:\> $procs = Get-Process
PS C:\> $procs | ForEach-Object { "The proc name is $($PSItem.Name)" }
The proc name is conhost
The proc name is csrss
The proc name is csrss
The proc name is dwm
The proc name is explorer
The proc name is Idle
The proc name is lsass
The proc name is MsMpEng
The proc name is powershell
The proc name is powershell_ise
The proc name is SearchIndexer
The proc name is services
The proc name is smss
The proc name is spoolsv
The proc name is svchost
```

These aren't Your Father's Commands

Always keep in mind that while PowerShell has things called **Dir** and **Cd**, they aren't the old MS-DOS commands. They're simply *aliases*, or nicknames, to PowerShell commands. That means they have different syntax.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". It displays two command executions:

```
PS C:\> dir /s
dir : Cannot find path 'C:\s' because it does not exist.
At line:1 char:1
+ dir /s
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (C:\s:String) [Get-ChildItem], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

PS C:\> dir -Recurse

Directory: C:\

Mode                LastWriteTime     Length Name
----                -              -         -
d---- 7/26/2012  12:33 AM        0 PerfLogs
d-r--  4/9/2013   11:02 PM        0 Program Files
d-r--  4/9/2013   11:03 PM        0 Program Files (x86)
d---- 3/5/2013   12:43 PM        0 reports
```

The first command, `dir /s`, fails because it cannot find the path 'C:\s'. The second command, `dir -Recurse`, successfully lists all subdirectories under the current directory (C:\).

You can run **help dir** (or ask for help on any other alias) to see the actual command name, and its proper syntax.