

THE BIG BOOK OF

# PowerShell Gotchas

Don Jones



**PowerShell.org**  
Resources, Answers, and Education



# **The Big Book of PowerShell Gotchas**

by Don Jones

Copyright ©2013 by Don Jones

Licensed under Creative Commons Attribution-NoDerivs 3.0 Unported

# Foreword

I answer a lot of questions in the PowerShell.org Forums - more than 1,000 posts so far - and over the years since PowerShell was released I've probably heard a dozen or so questions over and over and over. They're the "gotchas," the things nearly everyone trips up on now and again - and it's time to put them to bed.

That's what this guide will hopefully help accomplish. By putting all the gotchas in one place, maybe folks will find them before they run into them. Or at least, folks can refer people *here* to answer one gotcha they've encountered... and they'll find the rest of 'em here, too, and maybe avoid them.

This is a living guide, meaning I'll update it over time. Be sure to check back for new versions.

(Oh, and as a note - I used PowerShell v3 to write this Guide, but the principles are universal; the syntax may differ in other versions.)

And good luck.

Don Jones

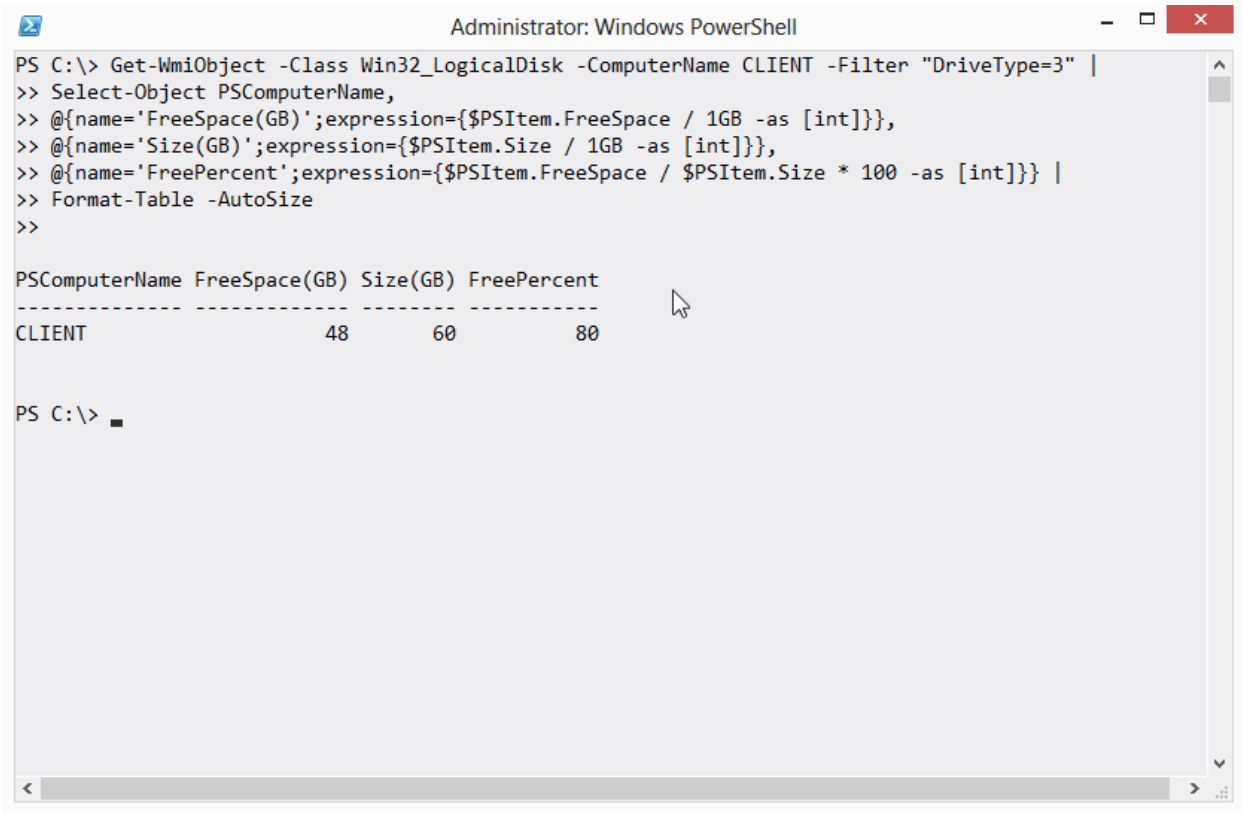
**Copyright** This guide is released under the Creative Commons Attribution-NoDerivs 3.0 Unported License. The authors encourage you to redistribute this file as widely as possible, but ask that you do not modify the document. However, you are encouraged to submit changes to the authors directly (<http://ConcentratedTech.com/contact>) so that your suggestions can be incorporated into the master document and published in a future revision.

# Contents

Format Right.....	4
-Contains isn't -Like .....	7
You Can't Have What You Don't Have .....	11
-Filter Values Diversity .....	15
Not Everything Produces Output.....	16
One HTML Page at a Time, Please .....	18
[Bloody] {Awful} (Punctuation) .....	20
Don't+Concatenate+Strings .....	22
\$ isn't Part of the Variable Name .....	24
Use the Pipeline, not an Array .....	27
Backtick, Grave Accent, Escape.....	29
A Crowd isn't an Individual .....	33
These aren't Your Father's Commands .....	35
Contacting Me .....	36

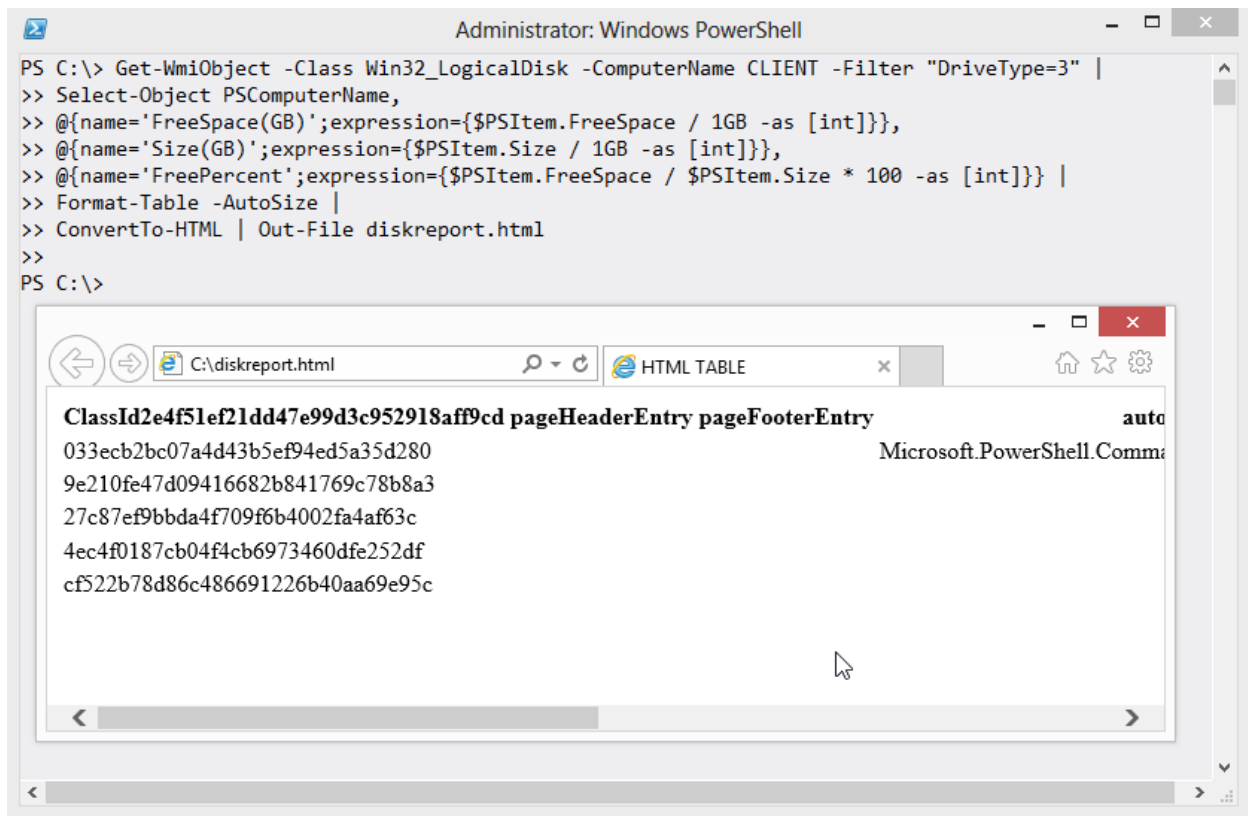
# Format Right

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



```
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  
>>  
  
PSComputerName FreeSpace(GB) Size(GB) FreePercent  
-----  
CLIENT                48         60         80  
  
PS C:\>
```

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



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:

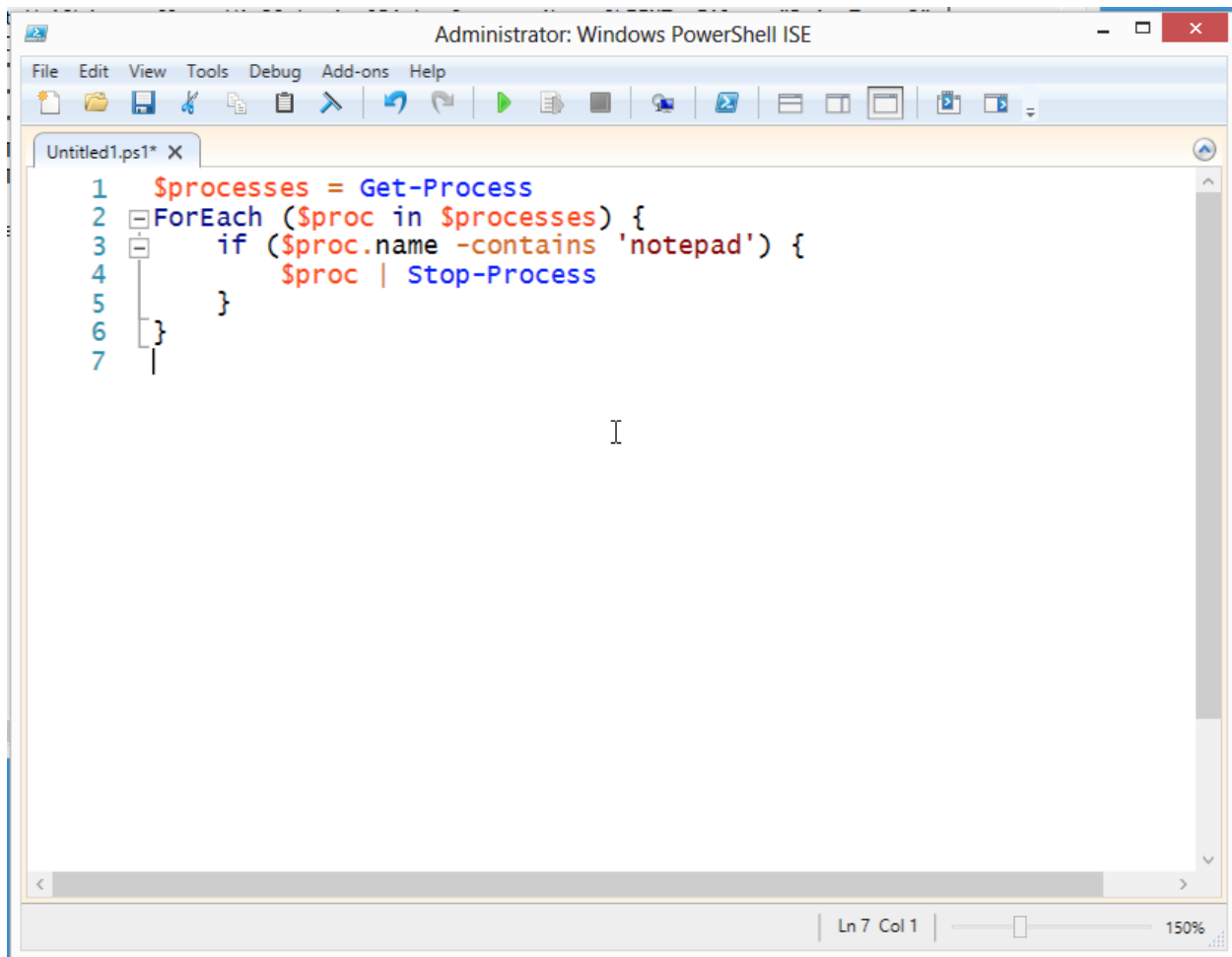
- Out-Default. This is technically always at the end of the pipeline, although it's invisible. It redirects to Out-Host.
- Out-Host also understands the output of Format commands, because Out-Host is how those formatting codes get on the screen in the first place.
- Out-Printer understands the formatting 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:

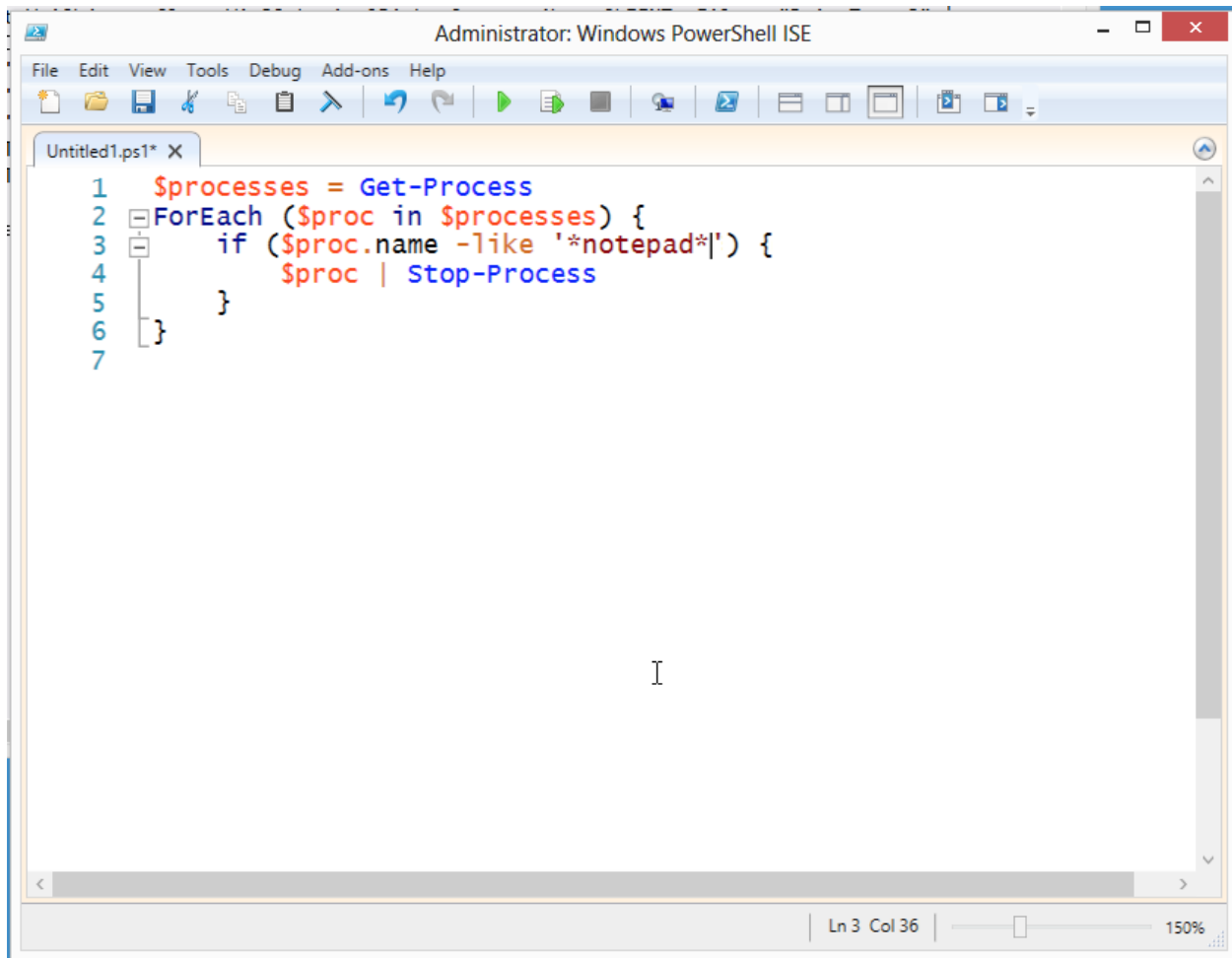
A screenshot of the Windows PowerShell ISE (Integrated Scripting Environment) window. The title bar reads "Administrator: Windows PowerShell ISE". The menu bar includes "File", "Edit", "View", "Tools", "Debug", "Add-ons", and "Help". The toolbar contains various icons for file operations, editing, and execution. The script editor shows a file named "Untitled1.ps1" with the following PowerShell code:

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

The code is color-coded: keywords are blue, variables and strings are red, and operators and punctuation are black. The status bar at the bottom indicates "Ln 7 Col 1" and a zoom level of "150%".

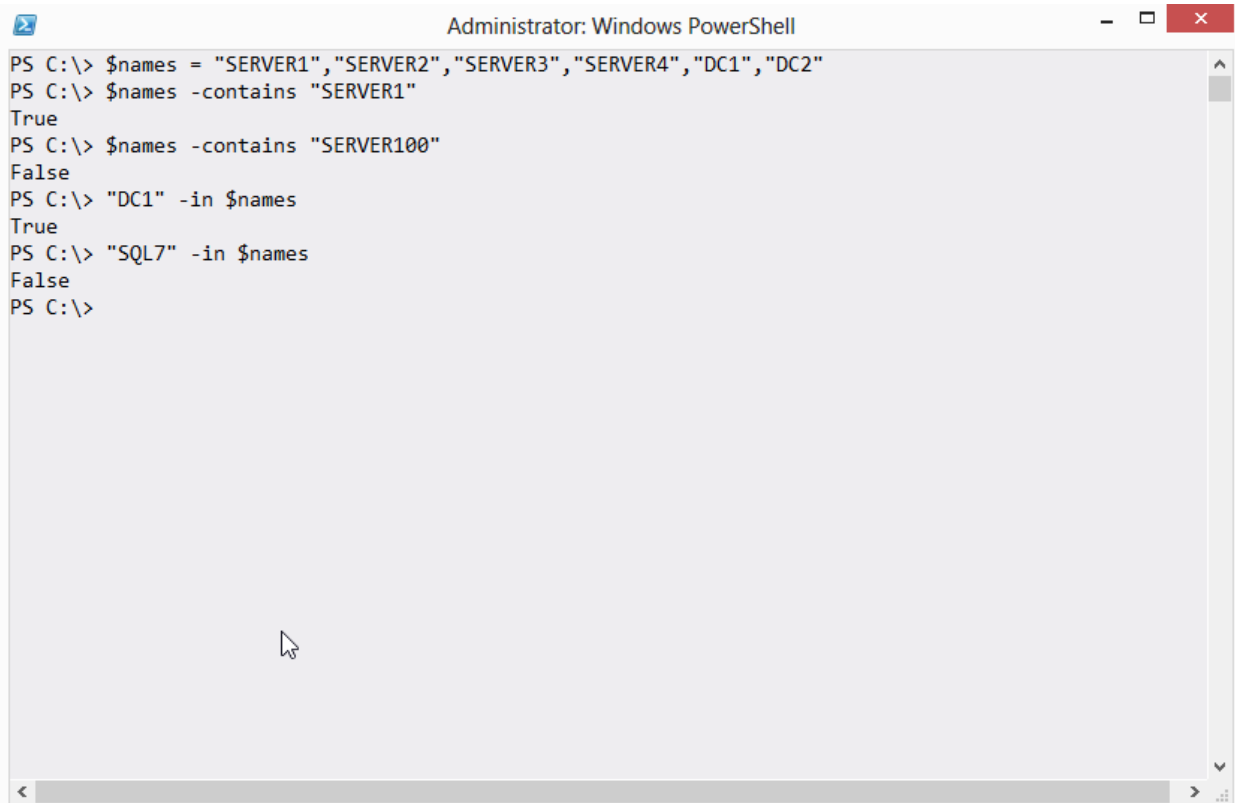
I get how this happens. The `-contains` operator *seems* like it should be checking to see if a process' name *contains* the letters "notepad." But that isn't what it does.

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



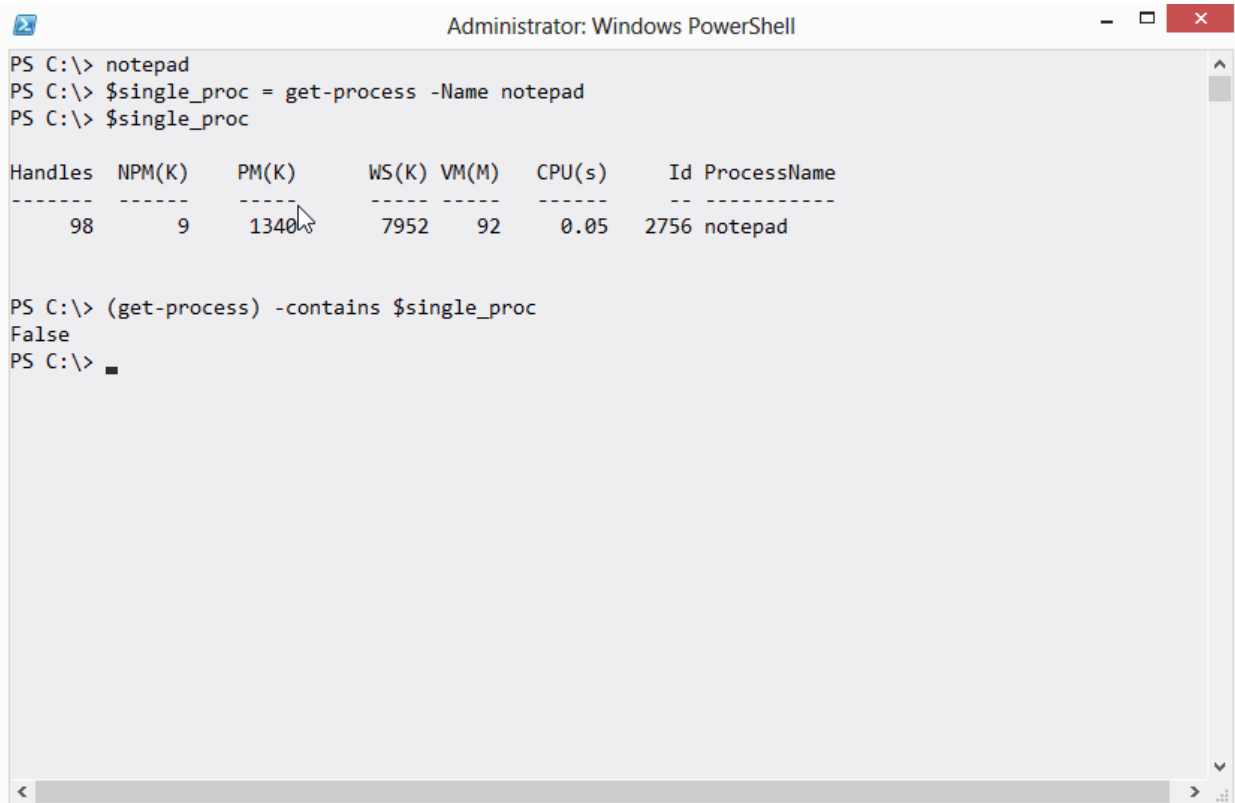
I'll let pass the thought that the *really correct* answer is to just run **Stop-Process -name \*notepad\***, because I was aiming for a simple example here. But... don't overthink things. Sometimes a script and a ForEach loop isn't the best approach.

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:



```
Administrator: Windows PowerShell
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.



```
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.



The best way to learn.

# Upgrade your brain

With advanced Microsoft® IT training from CBT Nuggets

Are you an Exchange admin, Active Directory architect, or SharePoint manager? Do you use Hyper-V? Do you struggle with storage and security, or worry about compliance and business continuity? Visit us at [cbtnuggets.com](http://cbtnuggets.com) for the latest advanced Microsoft IT training, and upgrade YOUR brain.

- Active Directory
- SharePoint
- SQL Server
- Hyper-V
- Programming
- Security
- System Center Config.
- Network Infrastructure
- Database
- Exchange Server
- PowerShell
- Lync Server
- Windows
- Windows Server
- Office
- Cloud
- Messaging
- Systems Management

Our nationally renowned trainers are the authority on Microsoft IT:



James Conrad



Don Jones  
*Microsoft MVP*



Greg Shields  
*Microsoft MVP*

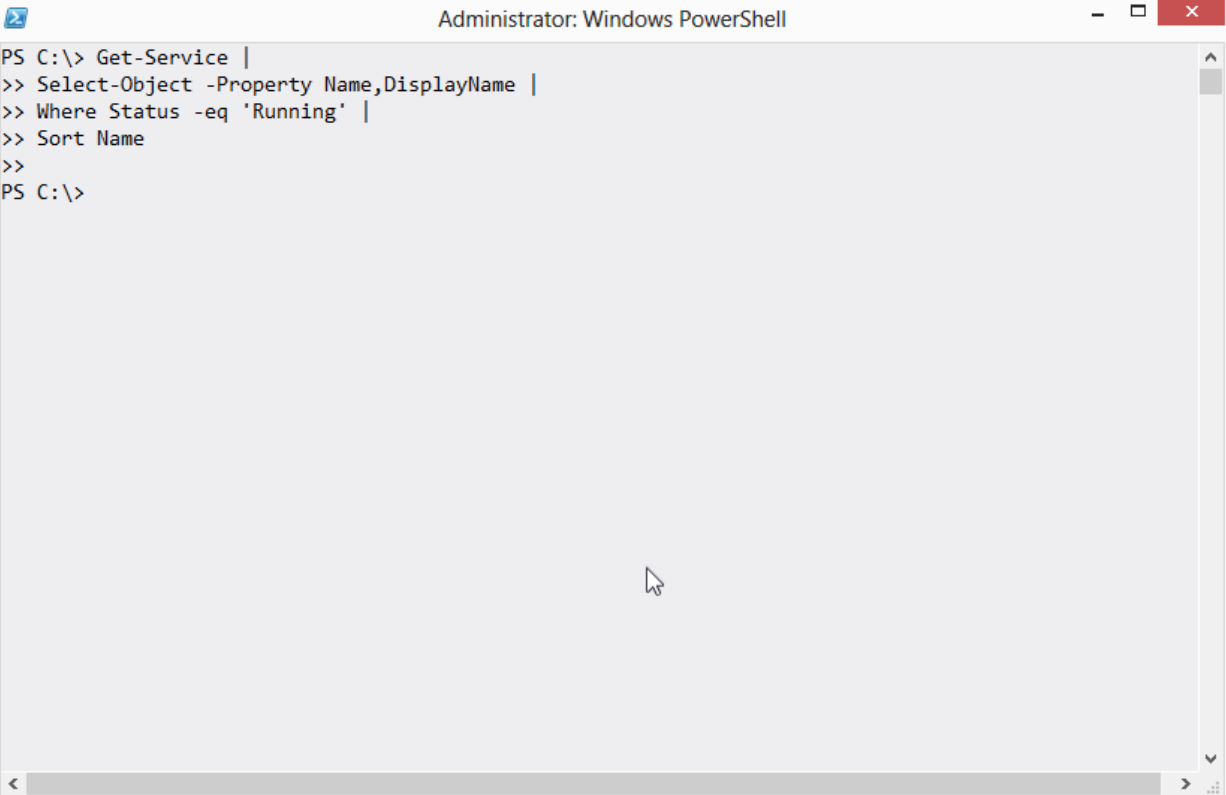


Tim Warner

**CBTNUGGETS.COM**

# You Can't Have What You Don't Have

Can you see what's wrong with this approach?



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

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command prompt shows a pipeline of commands: `Get-Service`, `Select-Object -Property Name,DisplayName`, `Where Status -eq 'Running'`, and `Sort Name`. The pipeline is not executed, as indicated by the prompt returning to `PS C:\>` without any output.

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.

```
Administrator: Windows PowerShell

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
>>

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.

```

wscsvc           Security Center
WSearch          Windows Search
WSService        Windows Store Service (WSService)
wuau servicing   Windows Update
wudfsvc          Windows Driver Foundation - User-mode Driver F...
WwanSvc          WWAN AutoConfig

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.

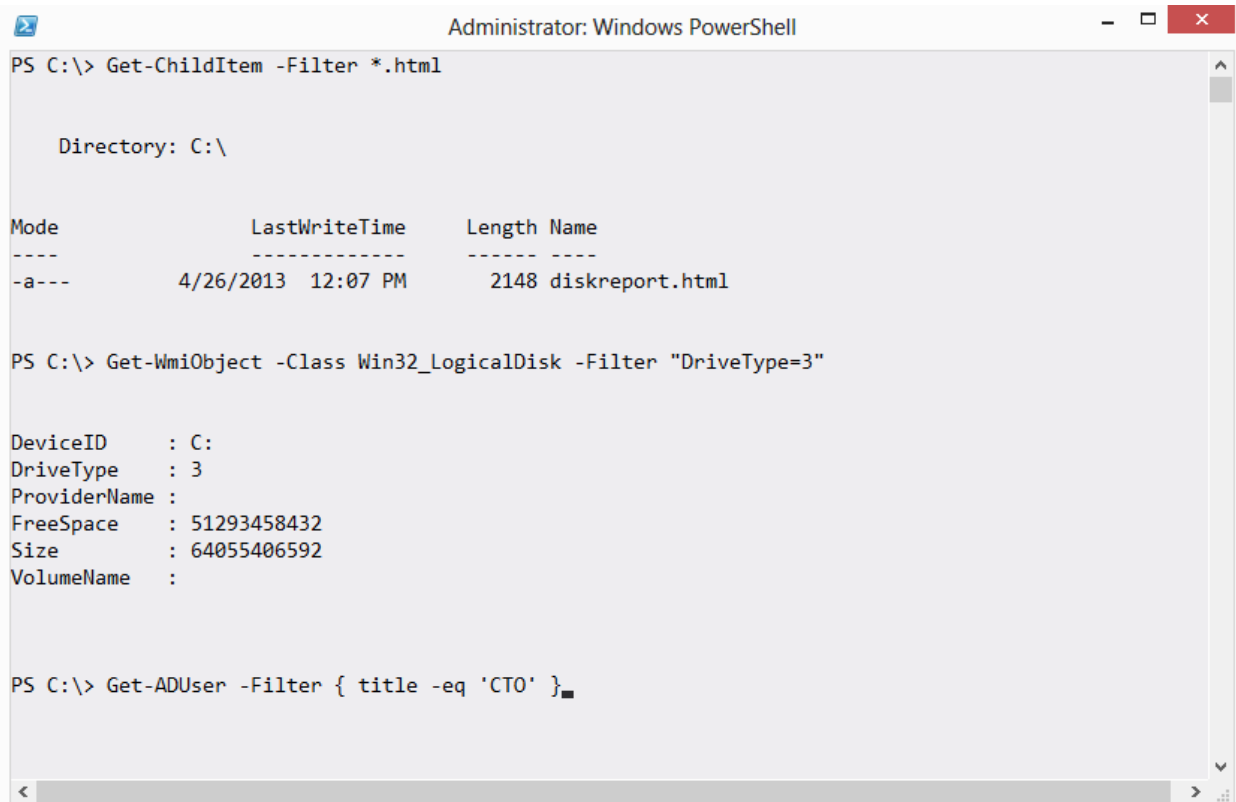
(Yeah, it'd be cooler if it threw an error - "Hey, you said to filter on the Status property, and there ain't one!" - but that isn't how it works.)

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:



```
Administrator: Windows PowerShell

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.

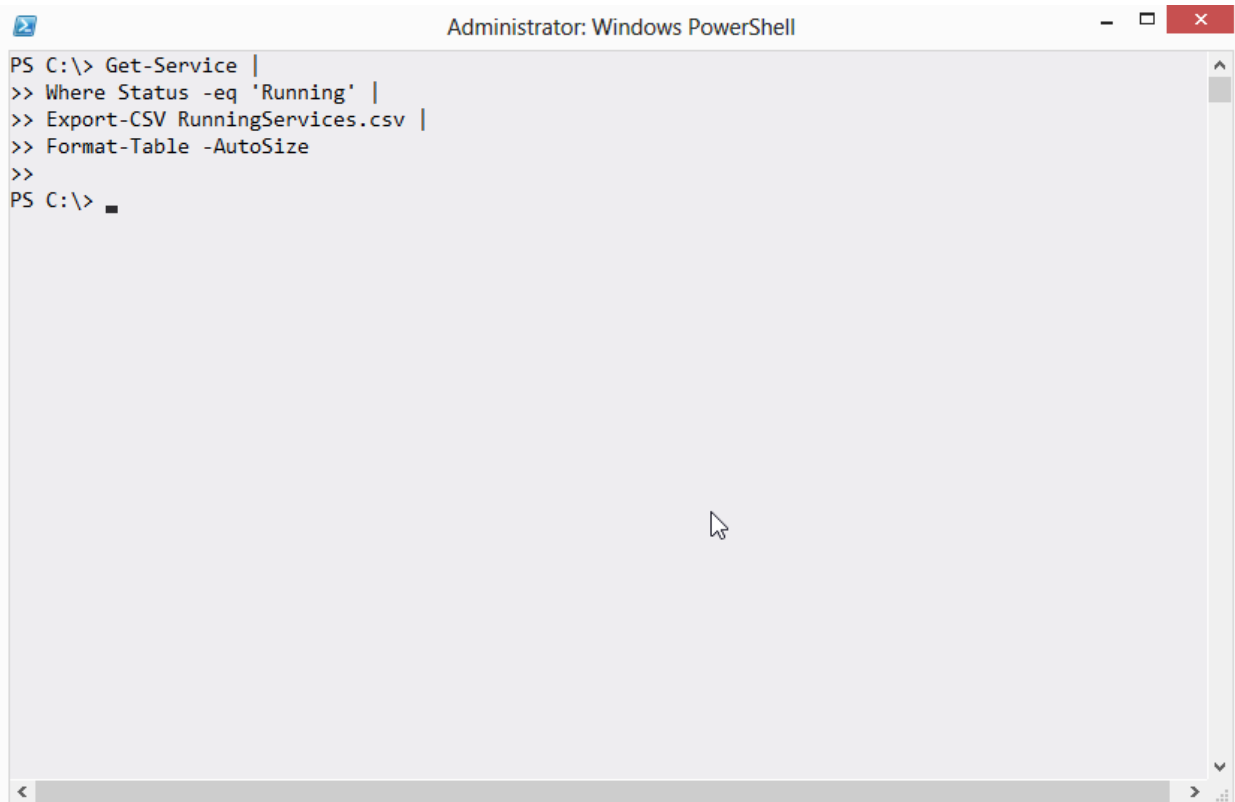
- With Get-ChildItem, -Filter accepts file system wildcards like \*.
- With Get-WmiObject, -Filter requires a string, and uses programming-style operators (like = for equality).
- With Get-ADUser, -Filter wanted a script block, and accepted PowerShell-style comparison operators (like -eq for equality).

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

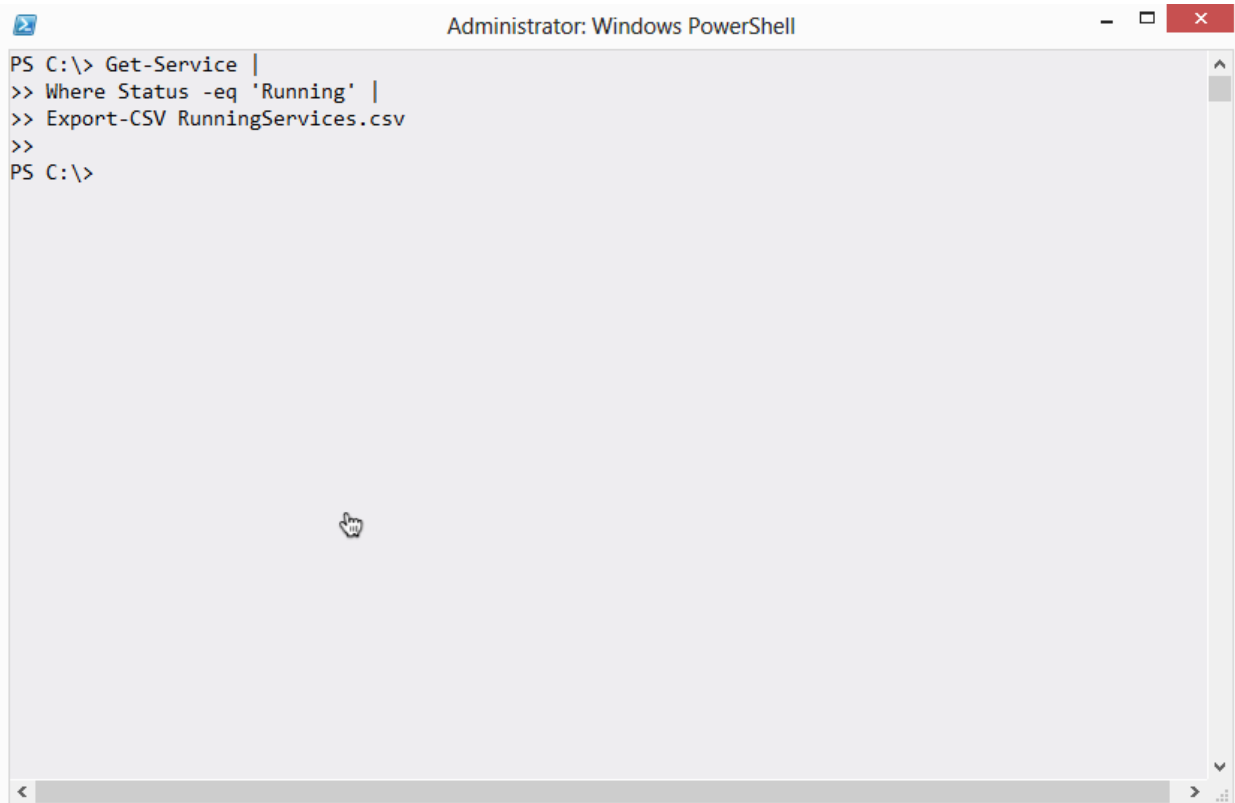
I see this one a lot in classes:



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

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 a Windows PowerShell window titled "Administrator: Windows PowerShell". The window has a standard Windows title bar with minimize, maximize, and close buttons. The command prompt shows the following text:

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

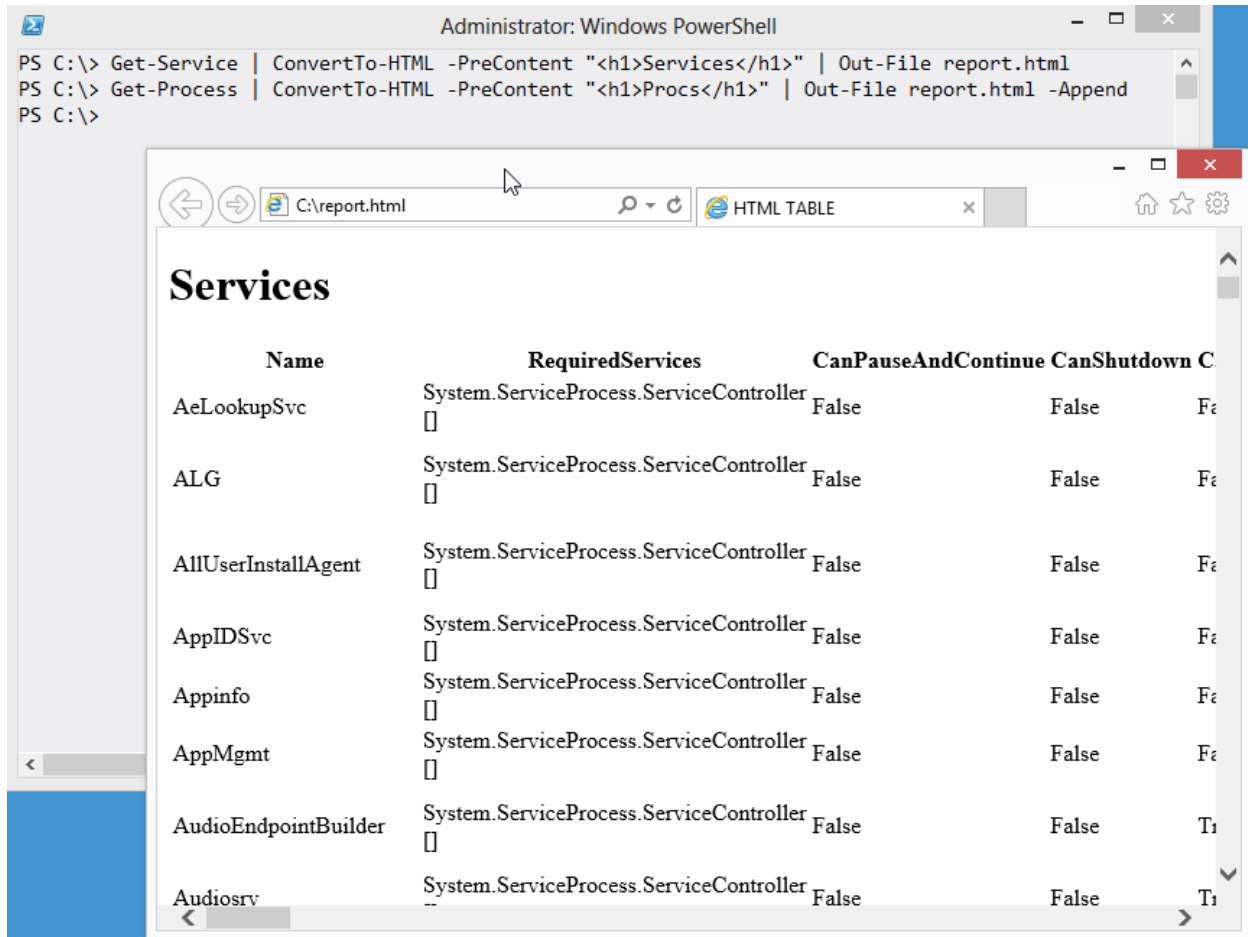
The cursor is at the end of the last line. The window is mostly empty, with a scrollbar on the right side.

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.

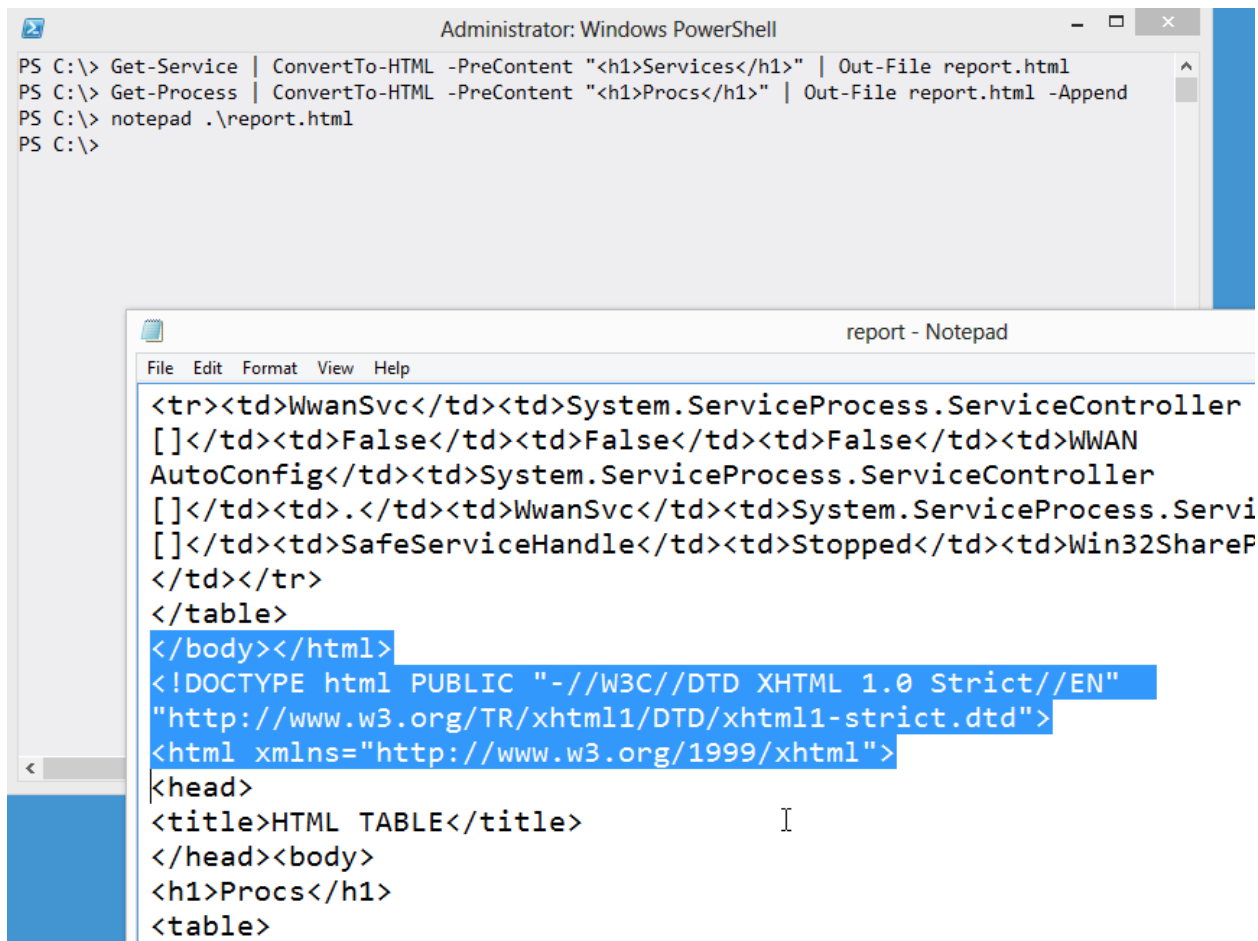
# One HTML Page at a Time, Please

This drives me batty:



What's happening is that someone ran two commands, 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:



The screenshot shows two overlapping windows. The top window is 'Administrator: Windows PowerShell' with the following commands and output:

```
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:\>
```

The bottom window is 'report - Notepad' showing the content of 'report.html'. The HTML is malformed, with the first page's closing tags appearing at the start of the second page's content:

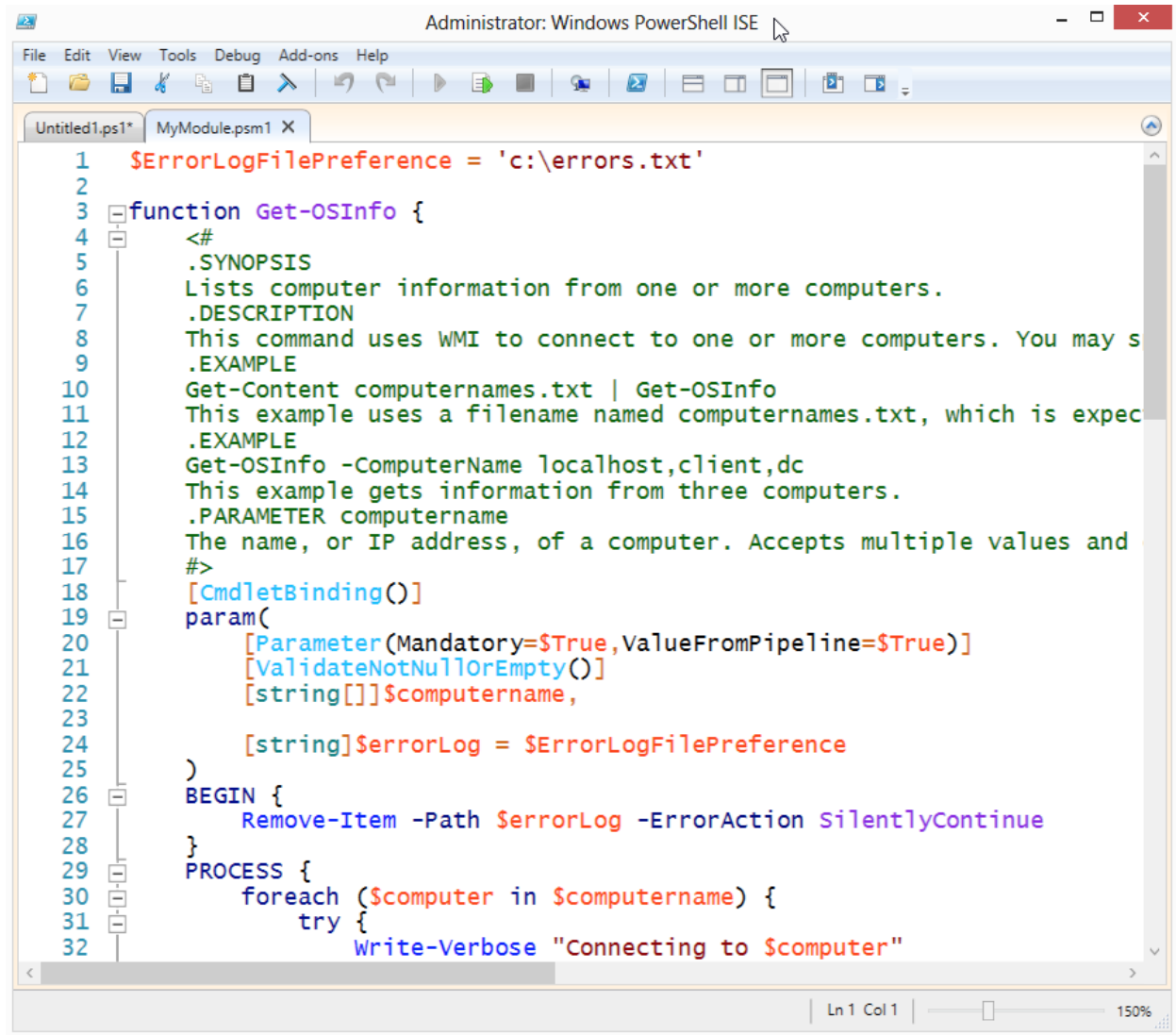
```
File Edit Format View Help
<tr><td>WwanSvc</td><td>System.ServiceProcess.ServiceController
[]</td><td>False</td><td>False</td><td>False</td><td>WWAN
AutoConfig</td><td>System.ServiceProcess.ServiceController
[]</td><td>.</td><td>WwanSvc</td><td>System.ServiceProcess.Servi
[]</td><td>SafeServiceHandle</td><td>Stopped</td><td>Win32ShareP
</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.



```
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.

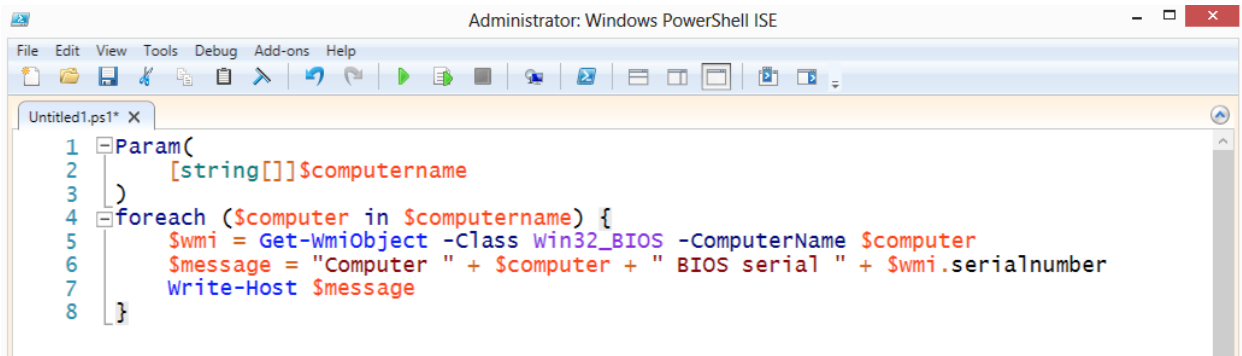
[Square brackets] are used around some attributes, like `[CmdletBinding()]`, and around data types like `[string]`, and to indicate arrays - as in `[string[]]`. They pop up a few other places, too.

{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.

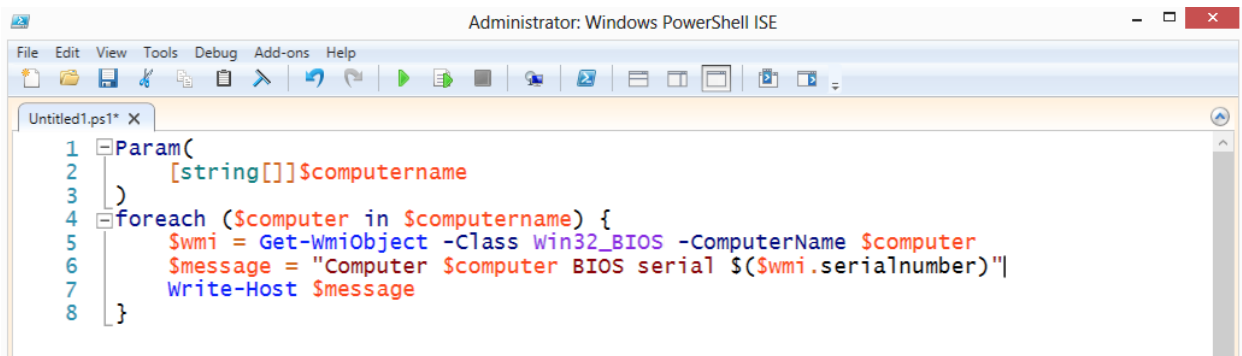
# Don't+Concatenate+Strings

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



```
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.



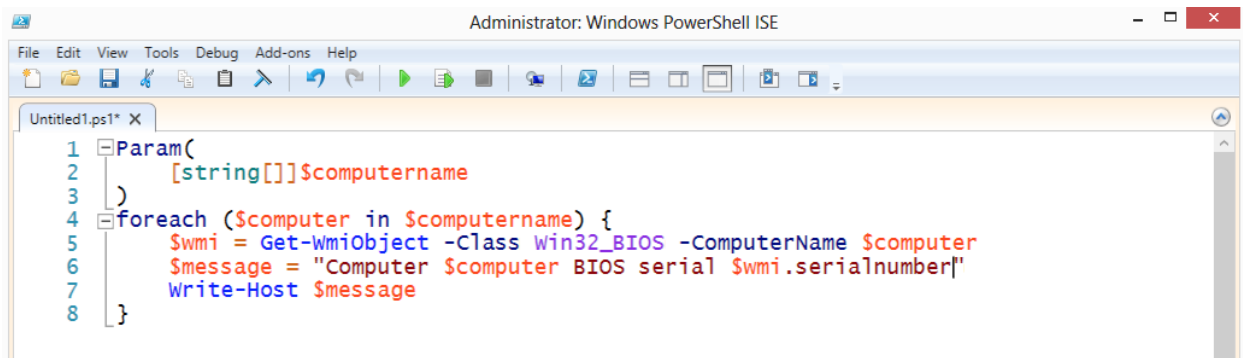
```
1 Param(  
2     [string[]]$computername  
3 )  
4 foreach ($computer in $computername) {  
5     $wmi = Get-WmiObject -Class Win32_BIOS -ComputerName $computer  
6     $message = "Computer $computer BIOS serial $($wmi.serialnumber)"  
7     Write-Host $message  
8 }
```

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**.

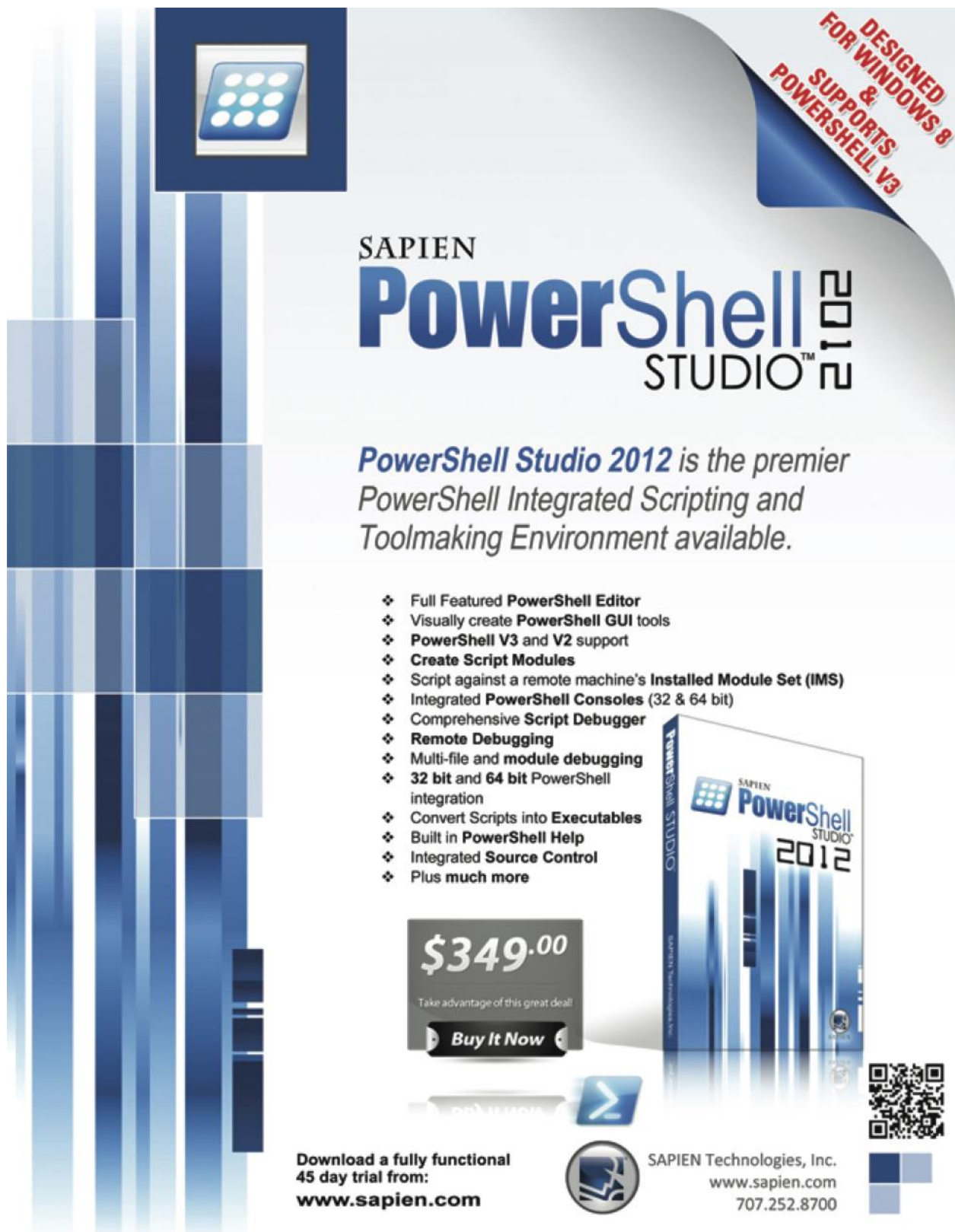
There's a sub-gotcha here:



A screenshot of the Windows PowerShell ISE (Integrated Scripting Environment) window. The title bar reads "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar contains various icons for file operations and execution. The script editor shows a file named "Untitled1.ps1" with the following PowerShell code:

```
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.



DESIGNED FOR WINDOWS 8  
SUPPORTS & POWERSHELL V3

# SAPIEN PowerShell STUDIO™


*PowerShell Studio 2012 is the premier PowerShell Integrated Scripting and Toolmaking Environment available.*

- ❖ Full Featured **PowerShell Editor**
- ❖ Visually create **PowerShell GUI** tools
- ❖ **PowerShell V3** and **V2** support
- ❖ **Create Script Modules**
- ❖ Script against a remote machine's **Installed Module Set (IMS)**
- ❖ Integrated **PowerShell Consoles** (32 & 64 bit)
- ❖ Comprehensive **Script Debugger**
- ❖ **Remote Debugging**
- ❖ Multi-file and module **debugging**
- ❖ **32 bit** and **64 bit** PowerShell integration
- ❖ Convert Scripts into **Executables**
- ❖ Built in **PowerShell Help**
- ❖ Integrated **Source Control**
- ❖ Plus much more

**\$349.00**  
Take advantage of this great deal  
**Buy It Now**

Download a fully functional 45 day trial from:  
**www.sapien.com**

SAPIEN Technologies, Inc.  
www.sapien.com  
707.252.8700



PrimalScript, SAPIEN PowerShell Studio, SAPIEN Software Suite (S3), iPowerShell Pro, ChangeVue, PrimalSQL, and PrimalXML are trademarks of SAPIEN Technologies, Inc. All other logos, trademarks, and service marks are the property of their respective owners. © 2002-2013 SAPIEN Technologies, Inc. All Rights Reserved.

# \$ 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:\>
```

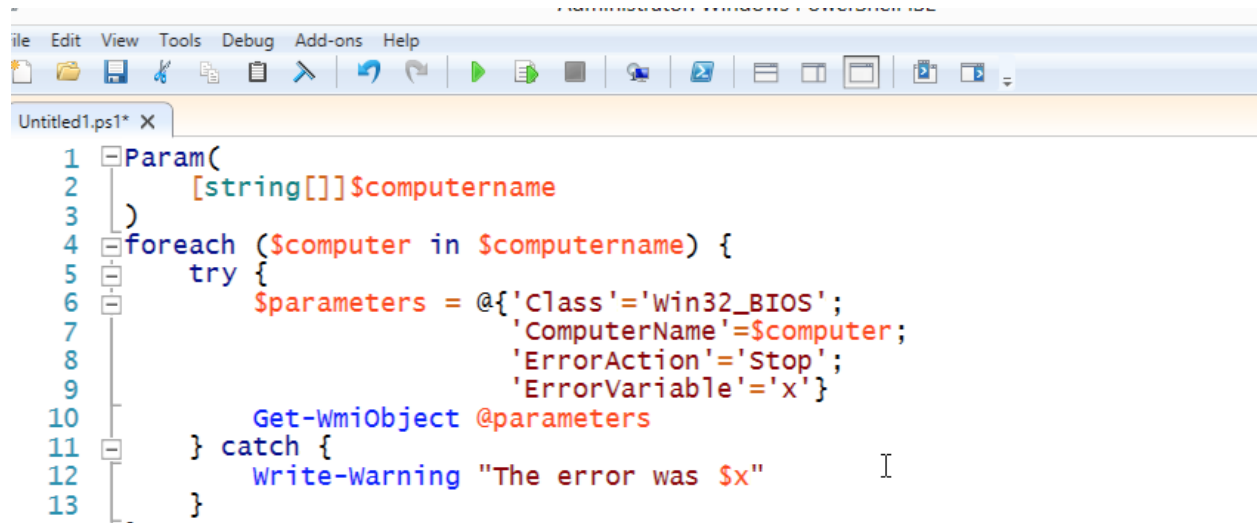
You see, the \$ is not part of the variable's name. If you have a variable named **example**, that's like having a box with "example" written on the side. Referring to **example** means you're talking *about the box itself*. Referring to **\$example** means you're messing with the *contents of the box*.

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:

```
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         $parameters = @{'Class'='Win32_BIOS';
7                         'ComputerName'=$computer;
8                         'ErrorAction'='Stop';
9                         'ErrorVariable'=$x}
10        Get-WmiObject @parameters
11    } catch {
12        Write-Warning "The error was $x"
13    }
```

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:

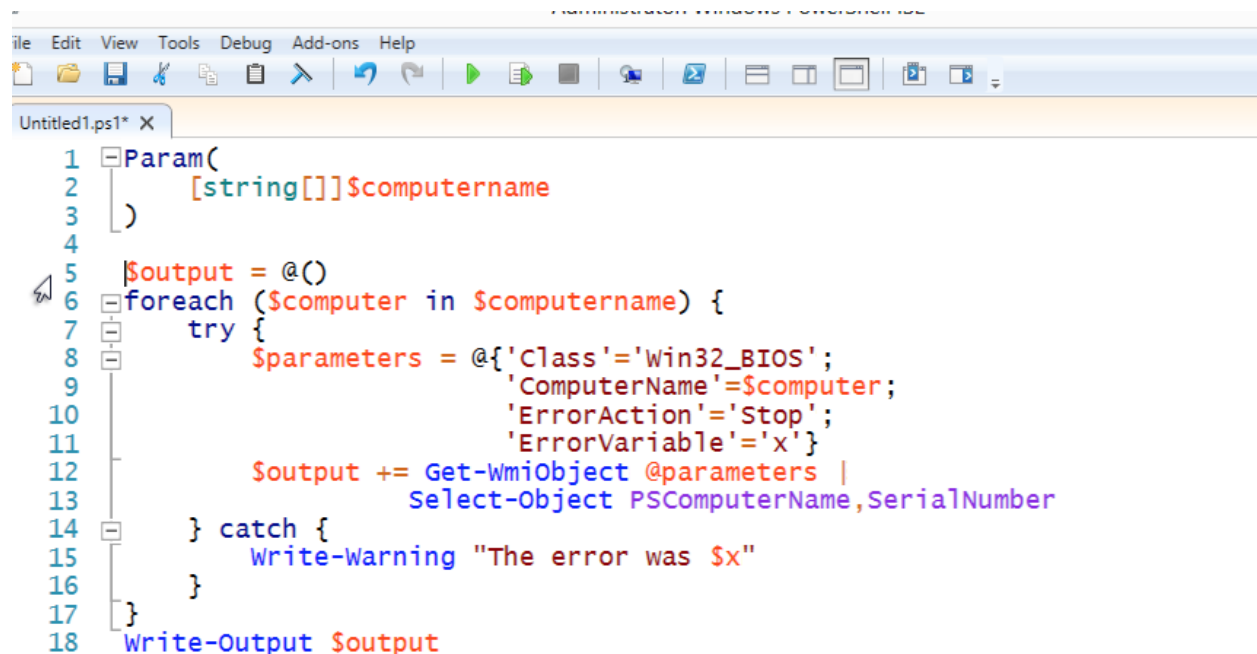


```
1 Param(  
2     [string[]]$computername  
3 )  
4 foreach ($computer in $computername) {  
5     try {  
6         $parameters = @{'Class'='Win32_BIOS';  
7                         'ComputerName'=$computer;  
8                         'ErrorAction'='Stop';  
9                         'ErrorVariable'='x'}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

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

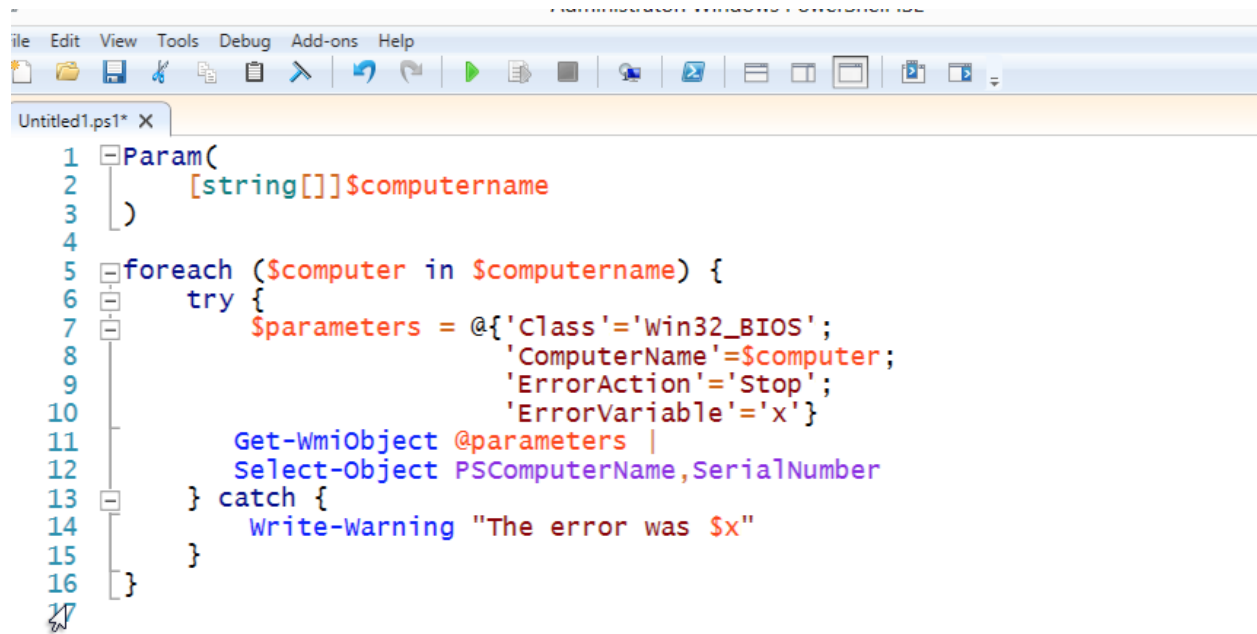
A very common mistake made by traditional programmers who come to PowerShell - which is *not* a programming language:

A screenshot of a PowerShell script editor window titled 'Untitled1.ps1'. The script is as follows:

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



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

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

# Try 7 days for FREE.



**Our video library is GROWING every week...**

SharePoint · IT Security · Windows Server 2012 · Network Fundamentals · Project Management  
Lync Server · Windows 7 · Windows 8 · PowerShell · ITIL · Exchange Server · SQL Server

**interface™**  
Video Training for IT Pros

**FREE 7-Day Pass!**

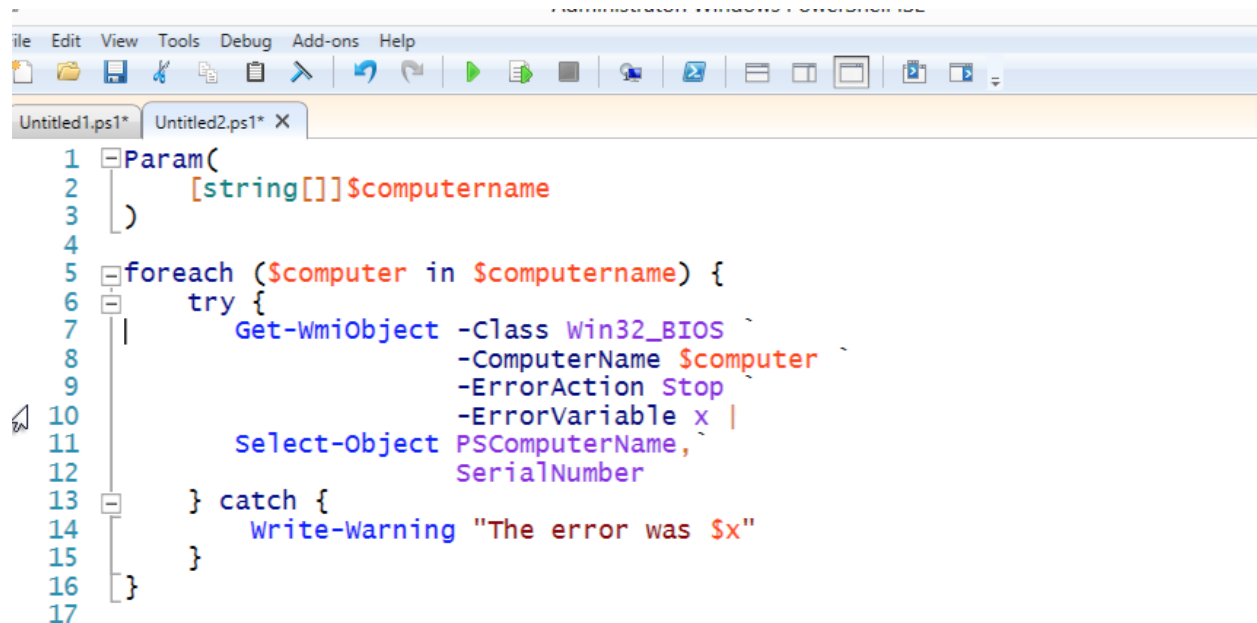
**Promo Code: PSR4AD**

Redeem at <http://videotraining.interfacett.com> | Expires 12/31/13



# Backtick, Grave Accent, Escape

You'll see folks do this a lot:

A screenshot of the PowerShell ISE (Integrated Scripting Environment) showing a script in a file named 'Untitled1.ps1'. The script 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 }
```

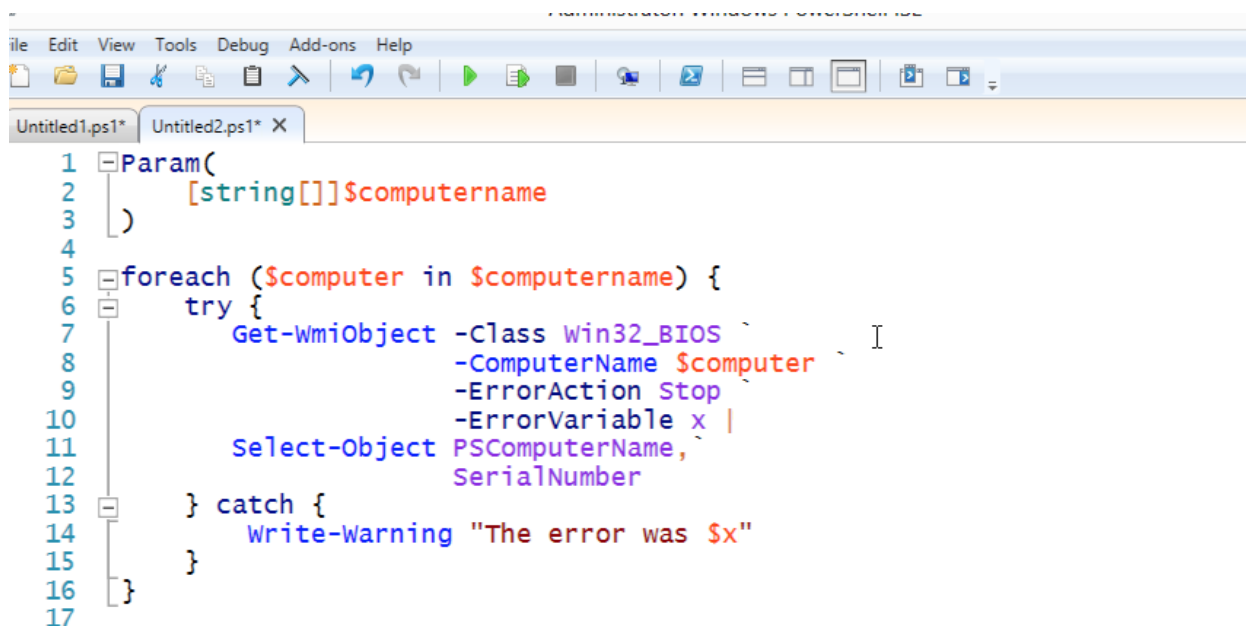
The script is syntactically correct. However, the author is pointing out a common mistake where a backtick is used at the end of a line to escape a carriage return, which is unnecessary and can be confusing.

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:





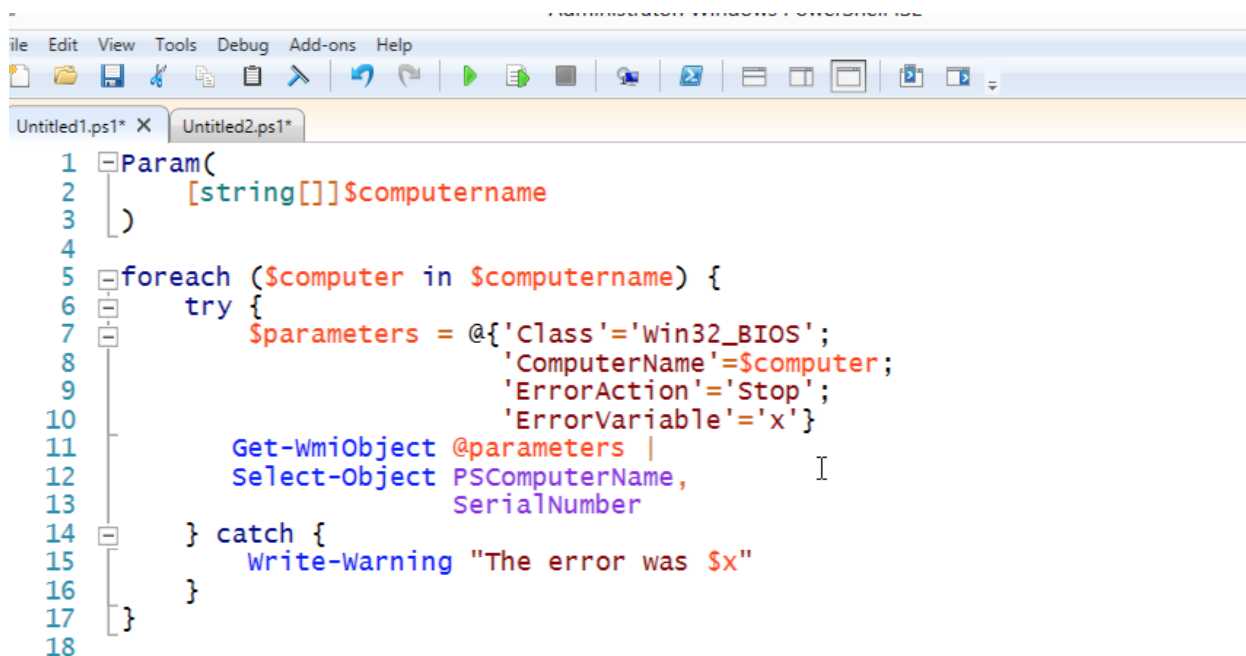
```
1 Param(
2     [string[]]$computername
3 )
4
5 foreach ($computer in $computername) {
6     try {
7         Get-WmiObject -Class Win32_BIOS `
8             -ComputerName $computer `
9             -ErrorAction Stop
10            -ErrorVariable x |
11        Select-Object PSComputerName,
12       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

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.



```
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    }
19 }
```

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.

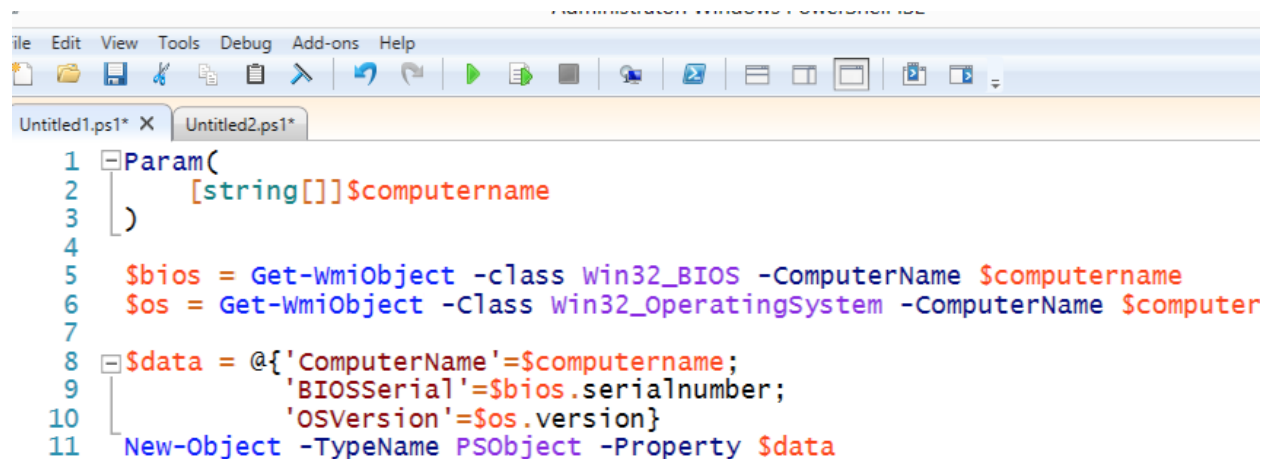
After Get-WmiObject, I have a pipe character - and you can legally line-break after that, too.

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.

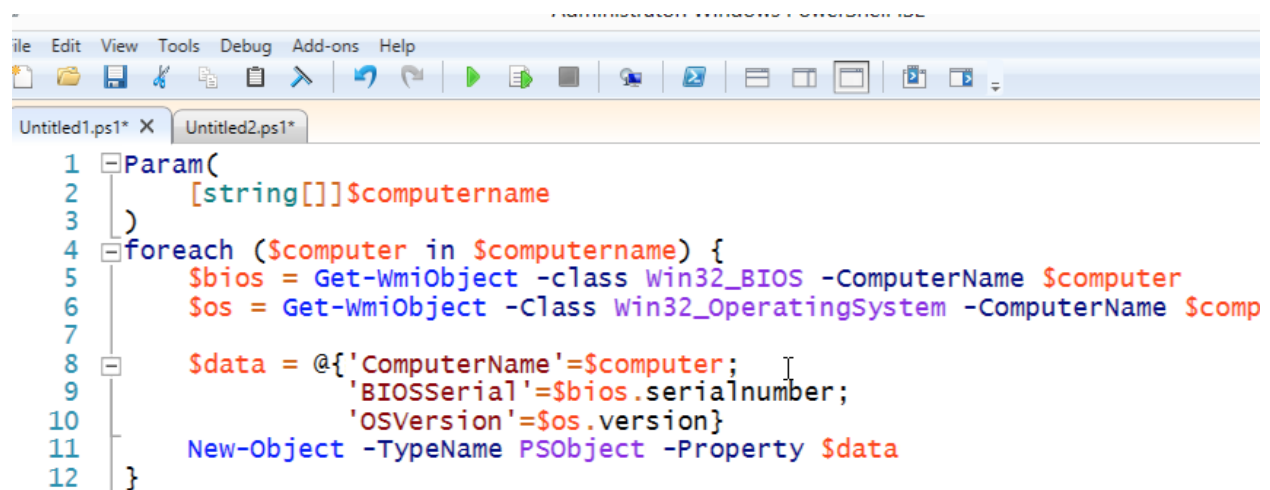
# A Crowd isn't an Individual

A very common newcomer mistake:



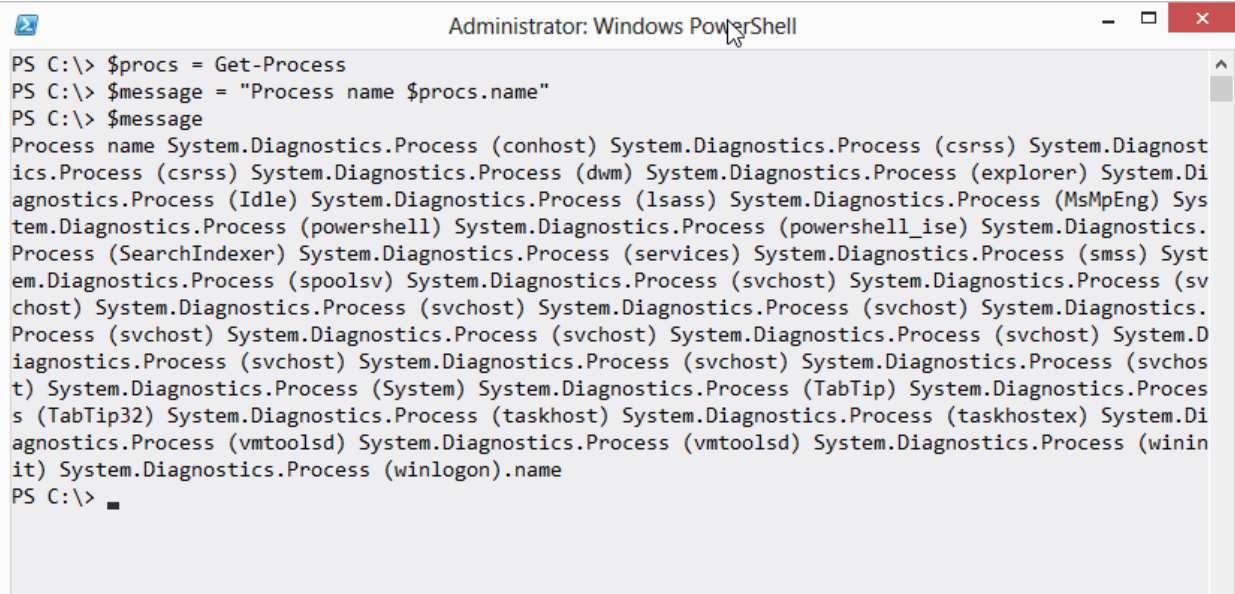
```
1 Param(  
2     [string[]]$computername  
3 )  
4  
5 $bios = Get-WmiObject -class Win32_BIOS -ComputerName $computername  
6 $os = Get-WmiObject -Class Win32_OperatingSystem -ComputerName $computer  
7  
8 $data = @{'ComputerName'=$computername;  
9           'BIOSerial'=$bios.serialnumber;  
10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

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



```

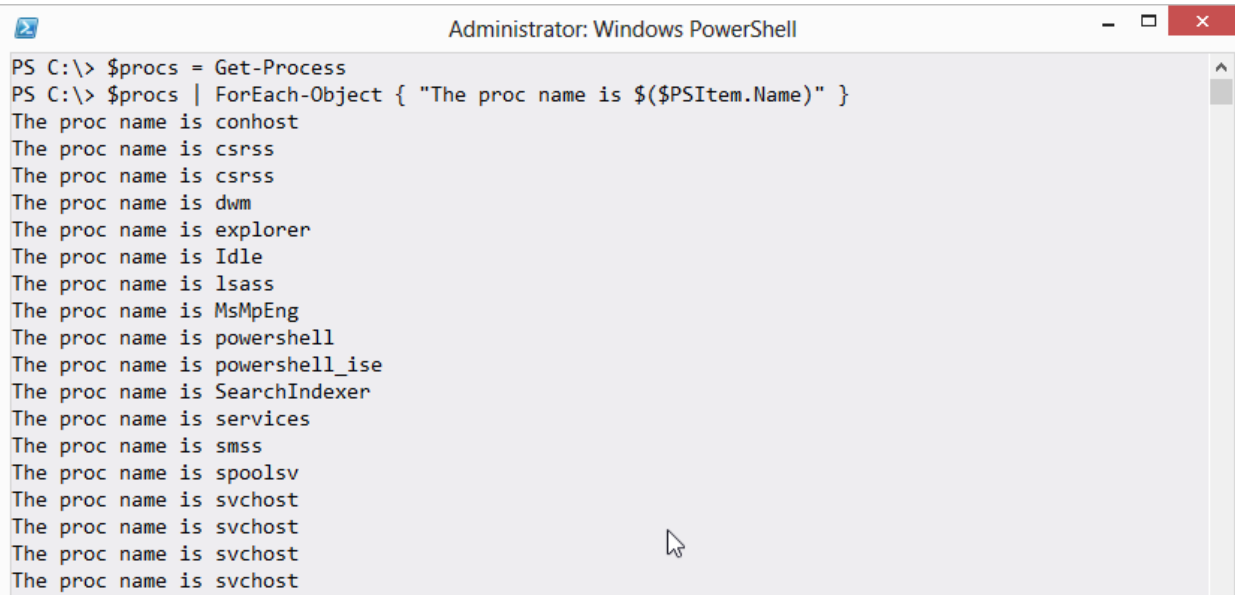
Administrator: Windows PowerShell

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.Diagnost
ics.Process (csrss) System.Diagnostics.Process (dwm) System.Diagnostics.Process (explorer) System.Di
agnostics.Process (Idle) System.Diagnostics.Process (lsass) System.Diagnostics.Process (MsMpEng) Sys
tem.Diagnostics.Process (powershell) System.Diagnostics.Process (powershell_ise) System.Diagnostics.
Process (SearchIndexer) System.Diagnostics.Process (services) System.Diagnostics.Process (smss) Syst
em.Diagnostics.Process (spoolsv) System.Diagnostics.Process (svchost) System.Diagnostics.Process (sv
chost) System.Diagnostics.Process (svchost) System.Diagnostics.Process (svchost) System.Diagnostics.
Process (svchost) System.Diagnostics.Process (svchost) System.Diagnostics.Process (svchost) System.D
iagnostics.Process (svchost) System.Diagnostics.Process (svchost) System.Diagnostics.Process (svchos
t) System.Diagnostics.Process (System) System.Diagnostics.Process (TabTip) System.Diagnostics.Proces
s (TabTip32) System.Diagnostics.Process (taskhost) System.Diagnostics.Process (taskhostex) System.Di
agnostics.Process (vmtoolsd) System.Diagnostics.Process (vmtoolsd) System.Diagnostics.Process (winin
it) 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:



```

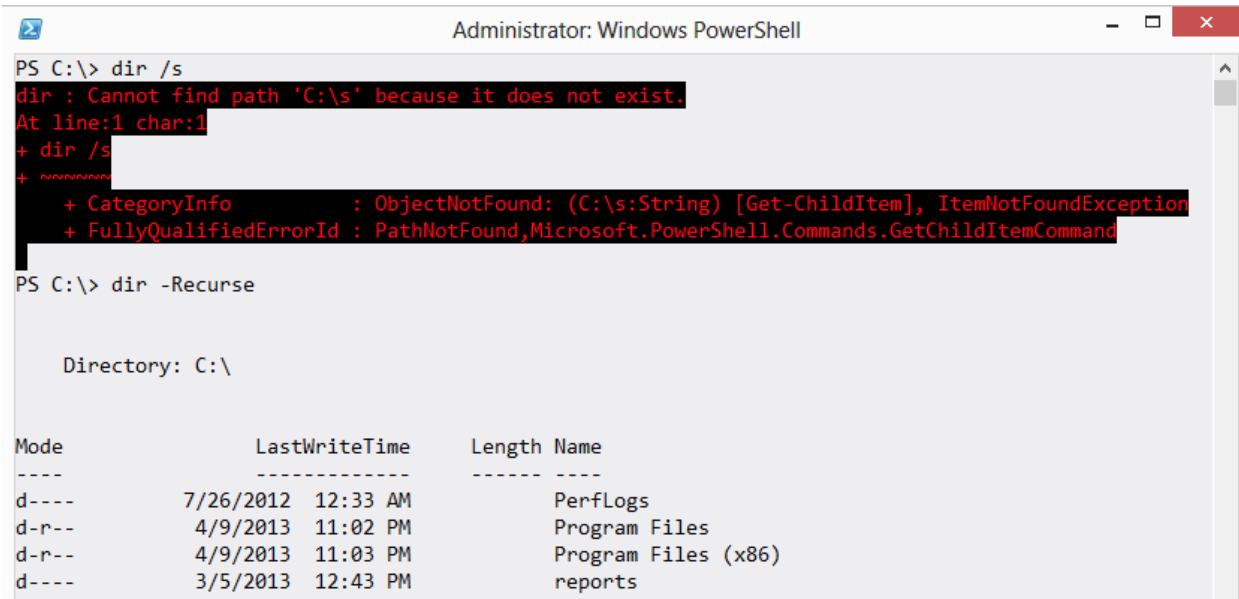
Administrator: Windows PowerShell

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
The proc name is svchost
The proc name is svchost
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.



```
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             PerfLogs
d-r--           4/9/2013   11:02 PM             Program Files
d-r--           4/9/2013   11:03 PM        Program Files (x86)
d-----          3/5/2013   12:43 PM             reports
```

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

## Contacting Me

If you're having problems, want to do something and can't figure out how, found a bug and want to offer a correction, or just have feedback on this guide or the EnhancedHTML module, I'd love to hear from you. The easiest way is to post in the "PowerShell Q&A" forum on <http://PowerShell.org/discuss>. I keep a pretty close eye on that, and I'll respond as soon as I'm able.

Do me a favor, if you post in the forums: Check out the Forum Tips. They're in the site's Site Info menu, right at the top of the page, and it's a very short post that will help you get the most from those forums.

Do check back on <http://PowerShellBooks.com> from time to time, to make sure you've got the most recent version of this guide and its code.