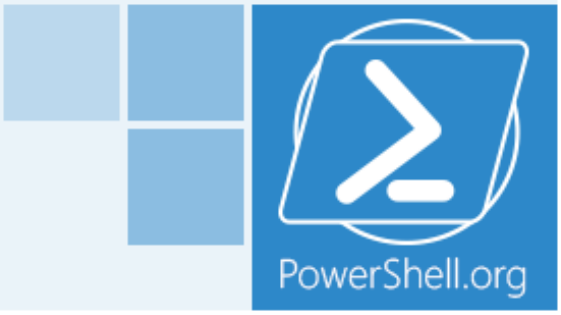
# The Big Book of PowerShell Gotchas

by Don Jones

curated by
Mike Shepard



# The Big Book of PowerShell Gotchas

By Don Jones

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

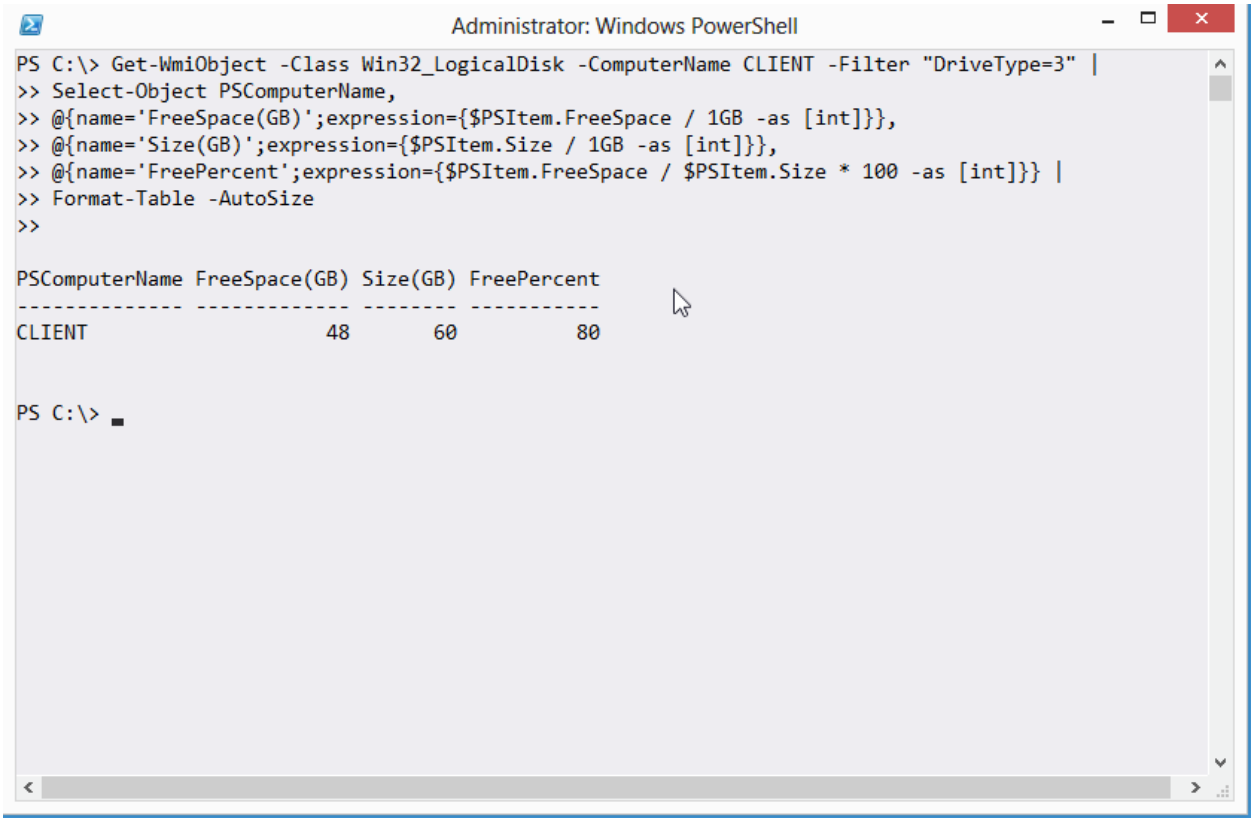



PowerShell.org ebooks are works-in-progress, and many are curated by members of the community. We encourage you to check back for new editions at least twice a year, by visiting PowerShell.org. You may also subscribe to our monthly e-mail TechLetter for notifications of updated ebook editions. Visit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

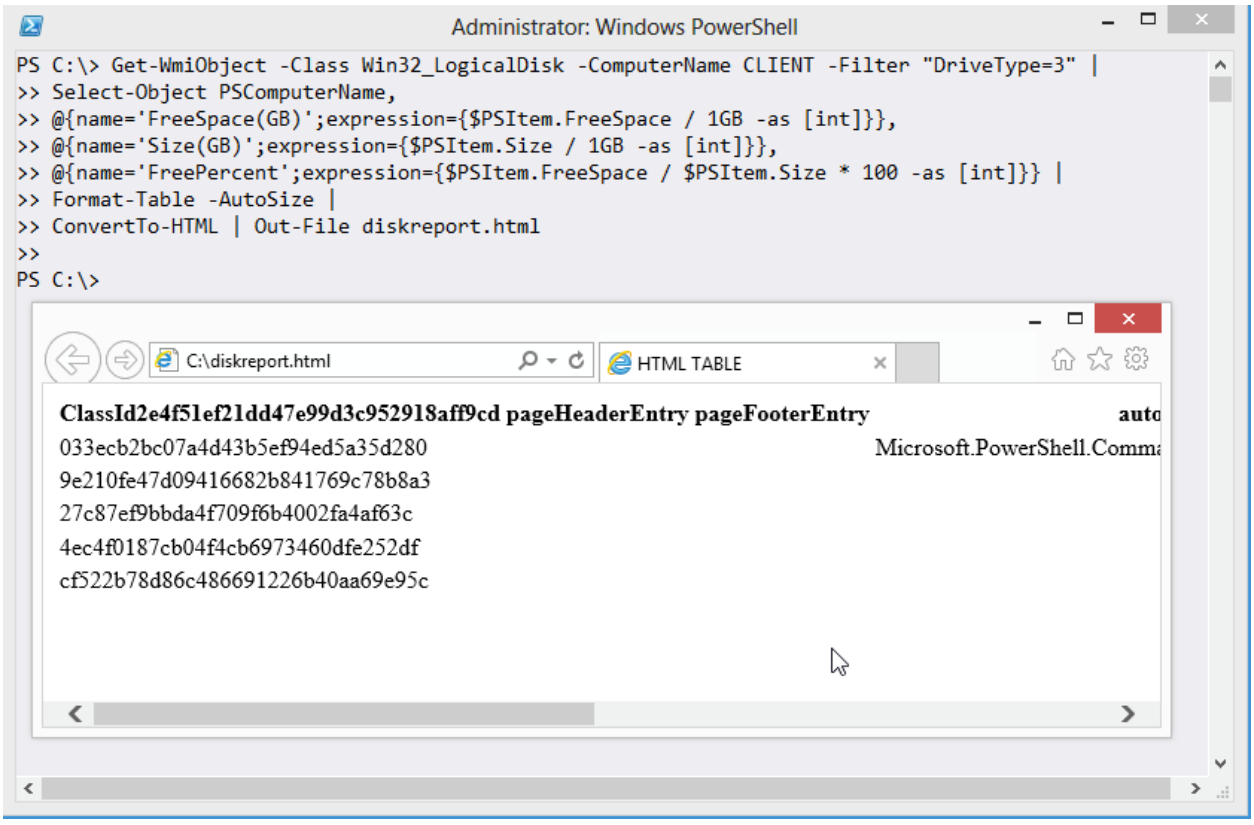
Feedback and corrections, as well as questions about this ebook's content, can be posted in the PowerShell Q&A forum on PowerShell.org. Moderators will make every attempt to either address your concern, or to engage the appropriate ebook author.

# Format Right

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




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.

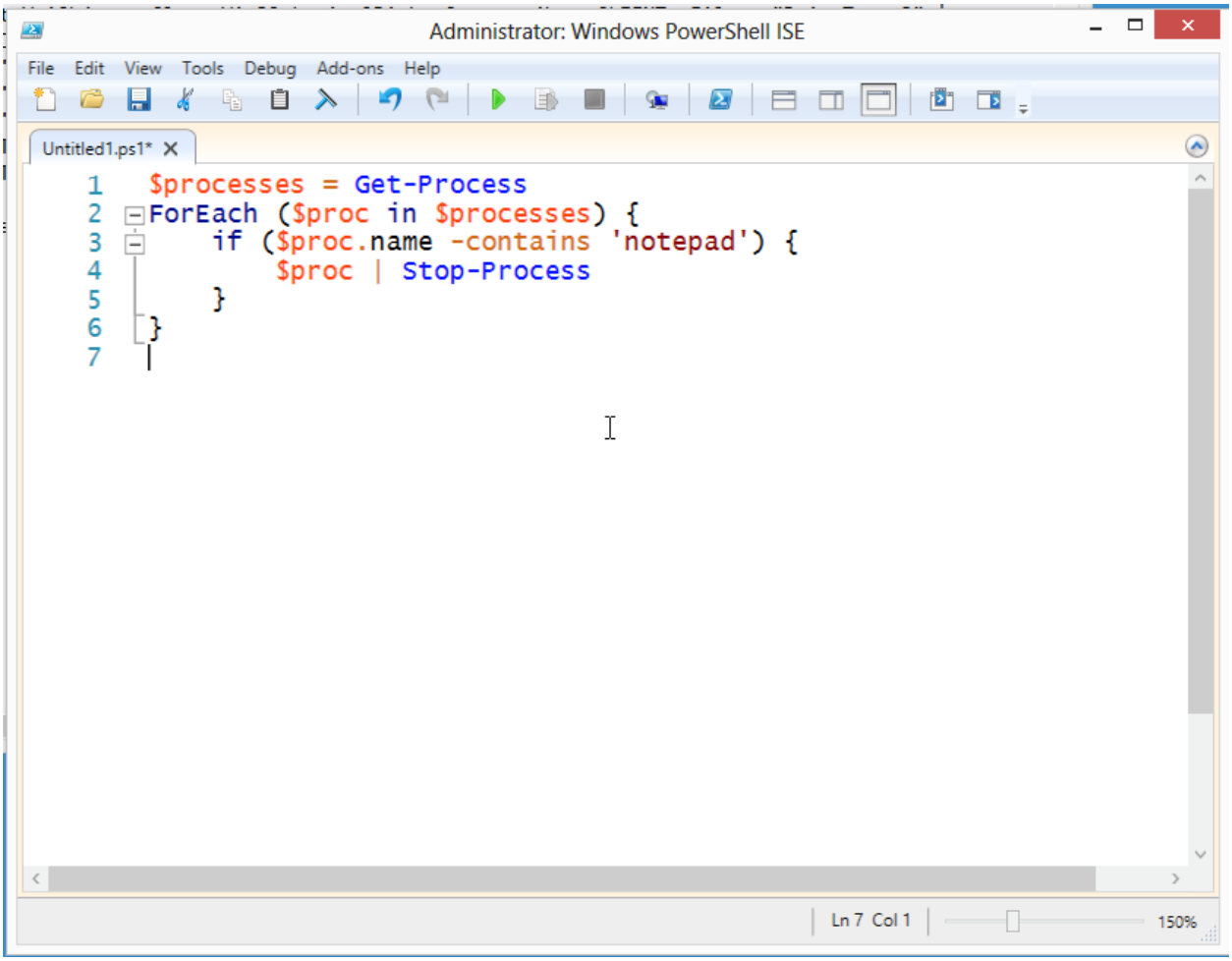
Out-File, like Out-Printer, redirects the on-screen output, but this time to a text file on disk.

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.

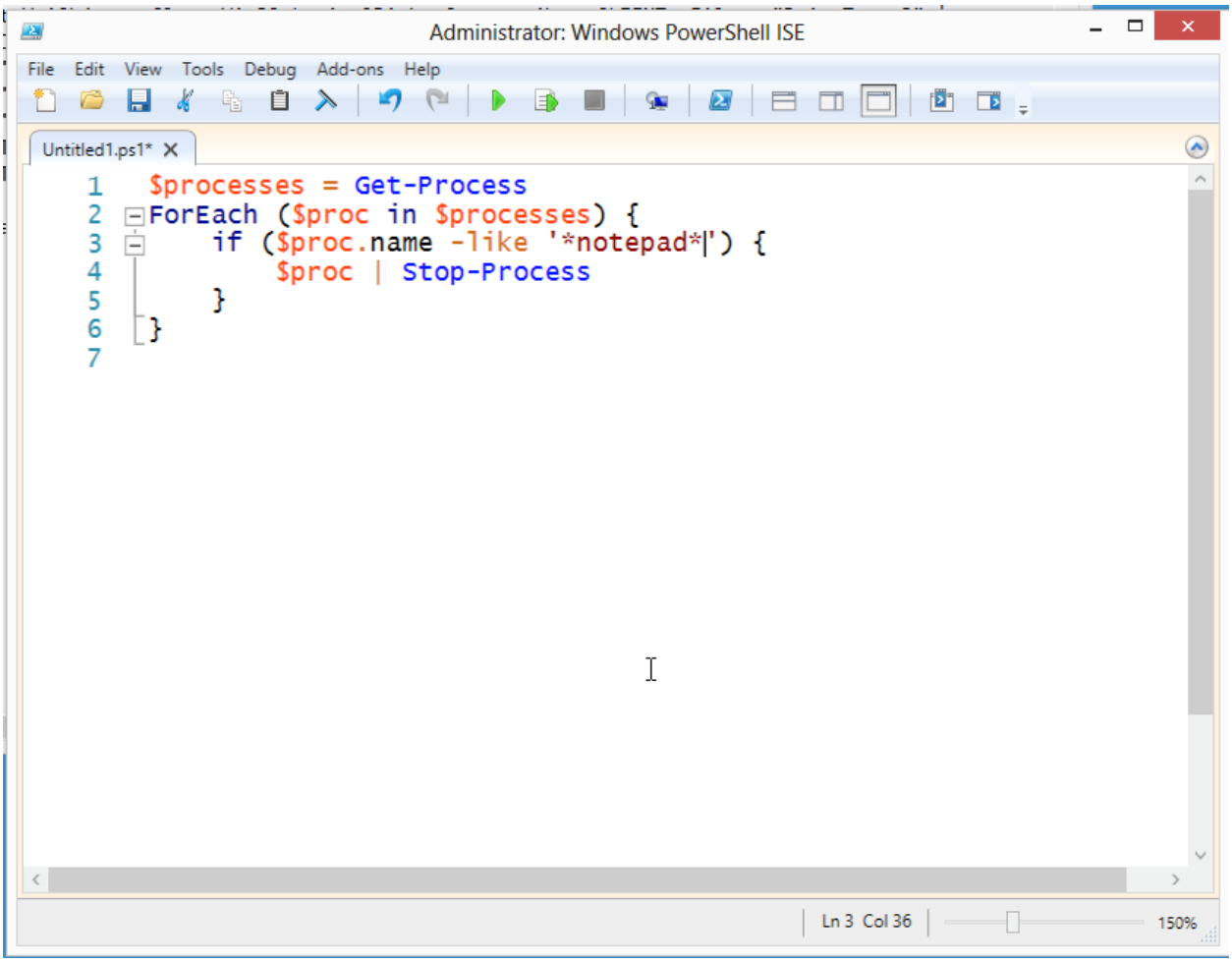
# -Contains isn't -Like

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



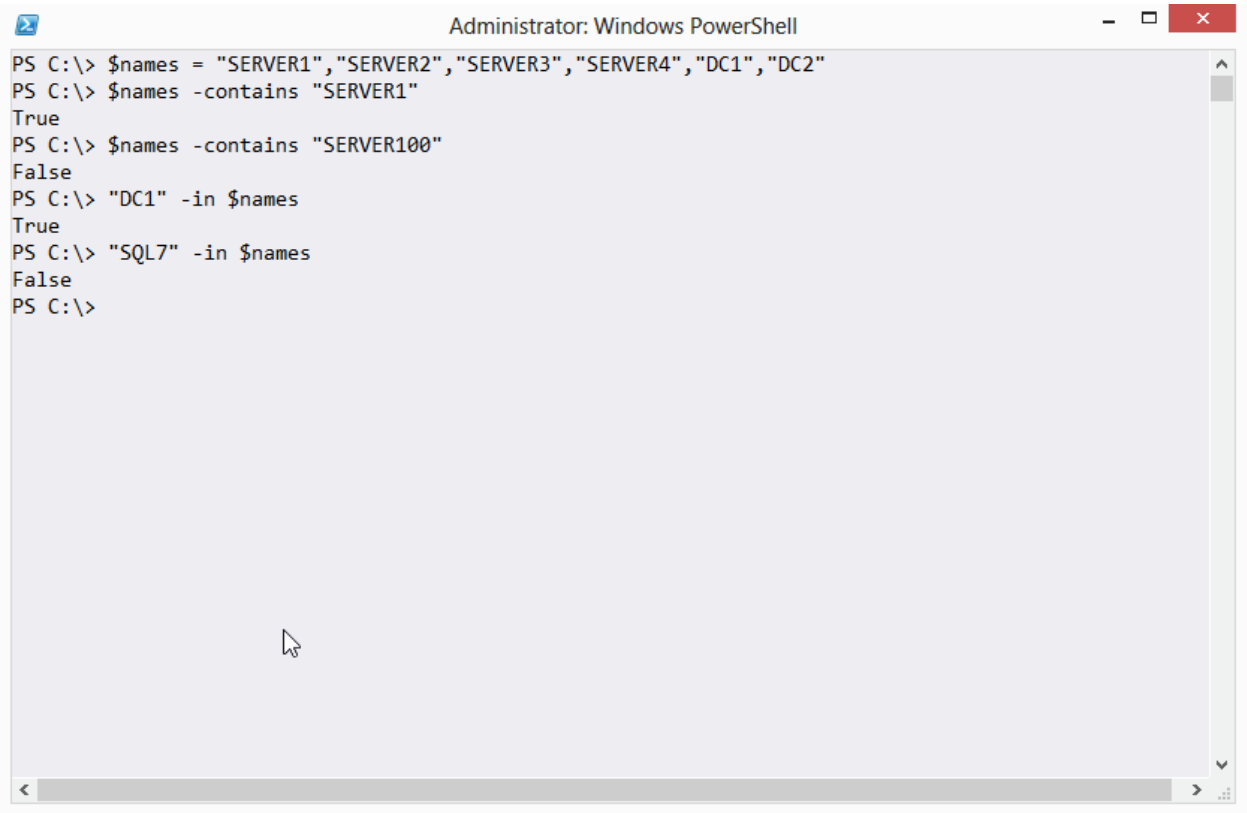

I get how this happens. The -contains operator *seems* like it should be checking to see if a process' name *contains* the letters "notepad." But that isn't what it does.

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

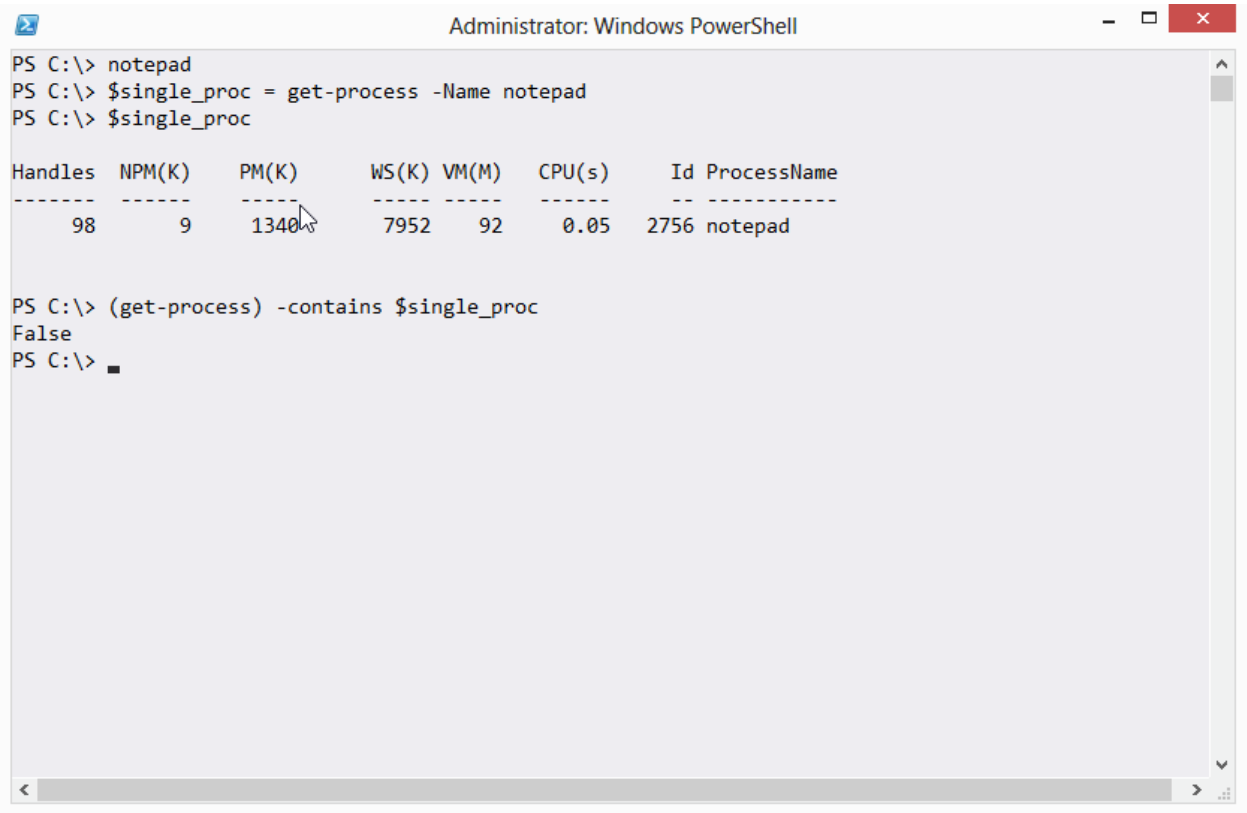



I'll let pass the thought that the *really correct* answer is to just run **Stop-Process -name *notepad***, because I was aiming for a simple example here. But... don't overthink things. Sometimes a script and a ForEach loop isn't the best approach.

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:




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.

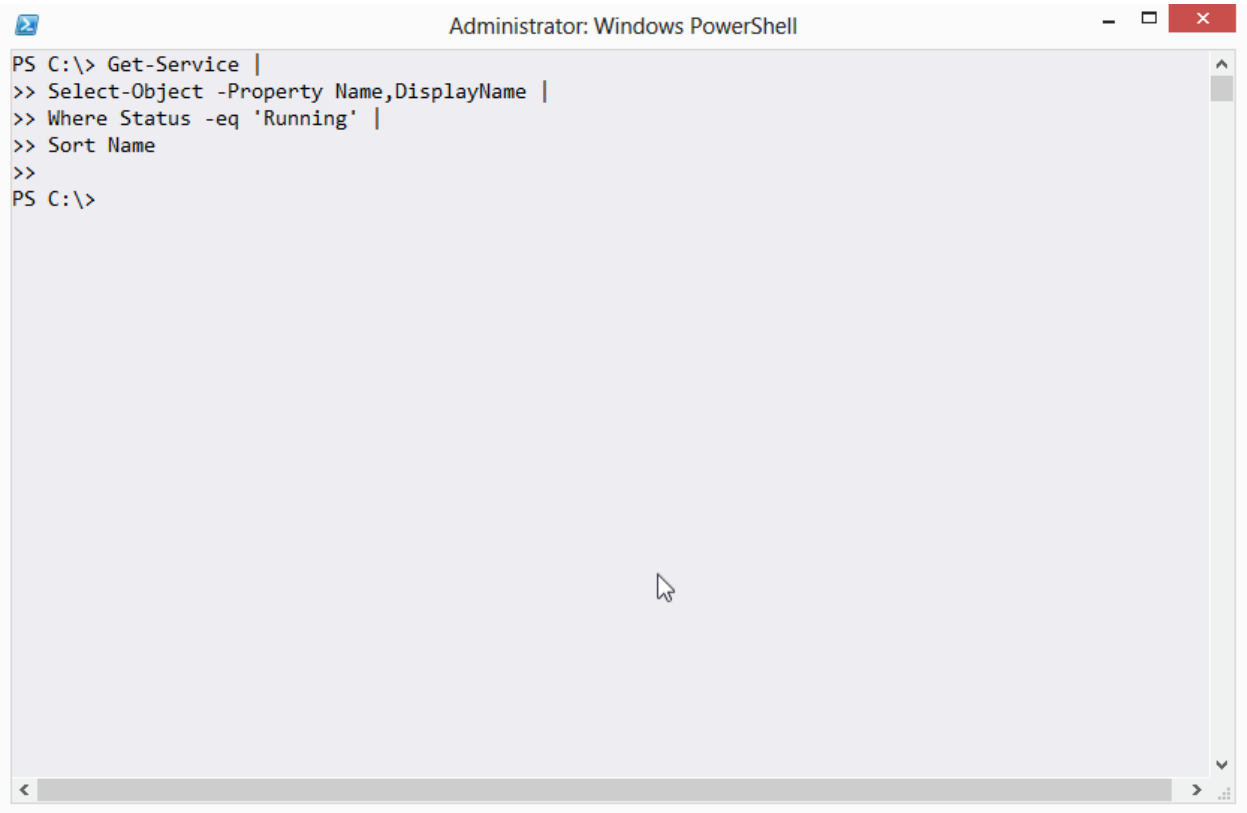



In this example, I've started Notepad. I've put its process object into $single_proc, and you can see that I verified it was there. But when I run Get-Process and check to see if its collection contained my Notepad, I got False. That's because the object in $single_proc is out of date. Notepad is running, 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.

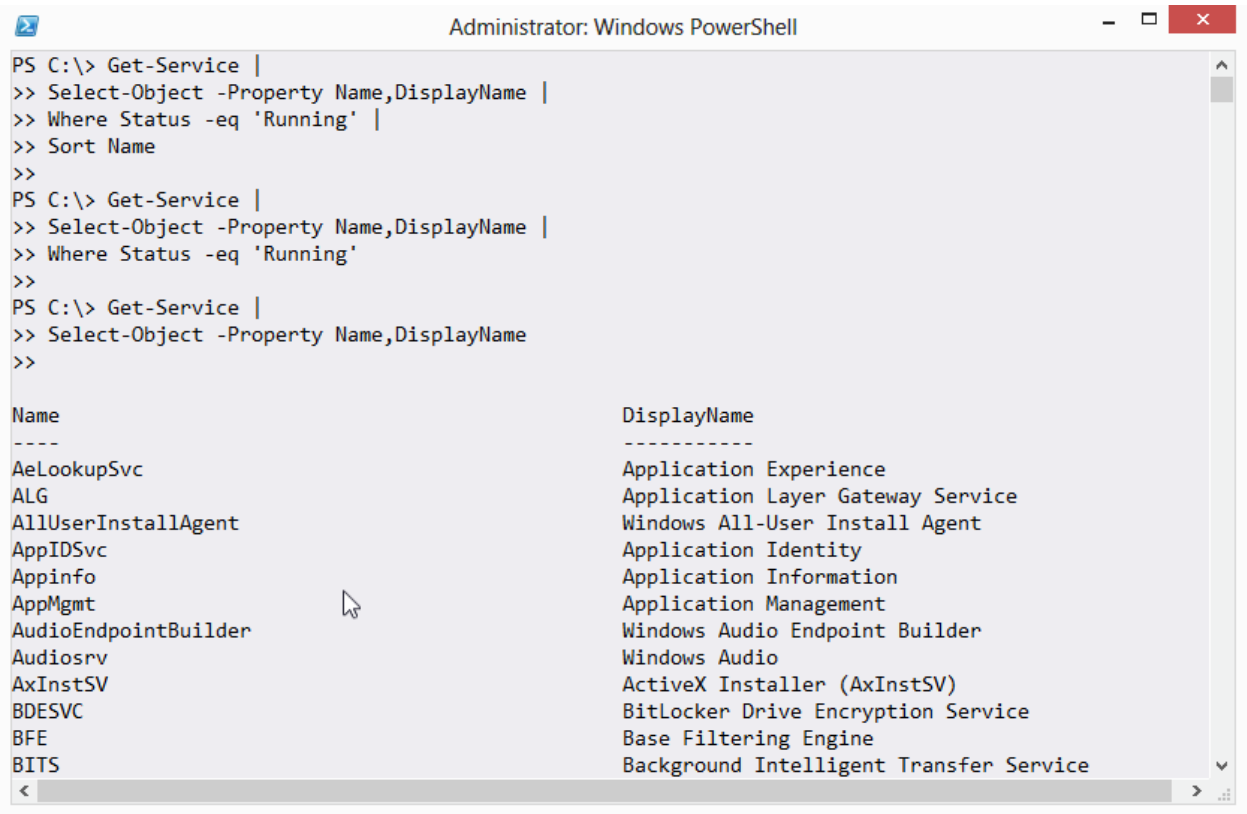
# You Can't Have What You Don't Have

Can you see what's wrong with this approach?

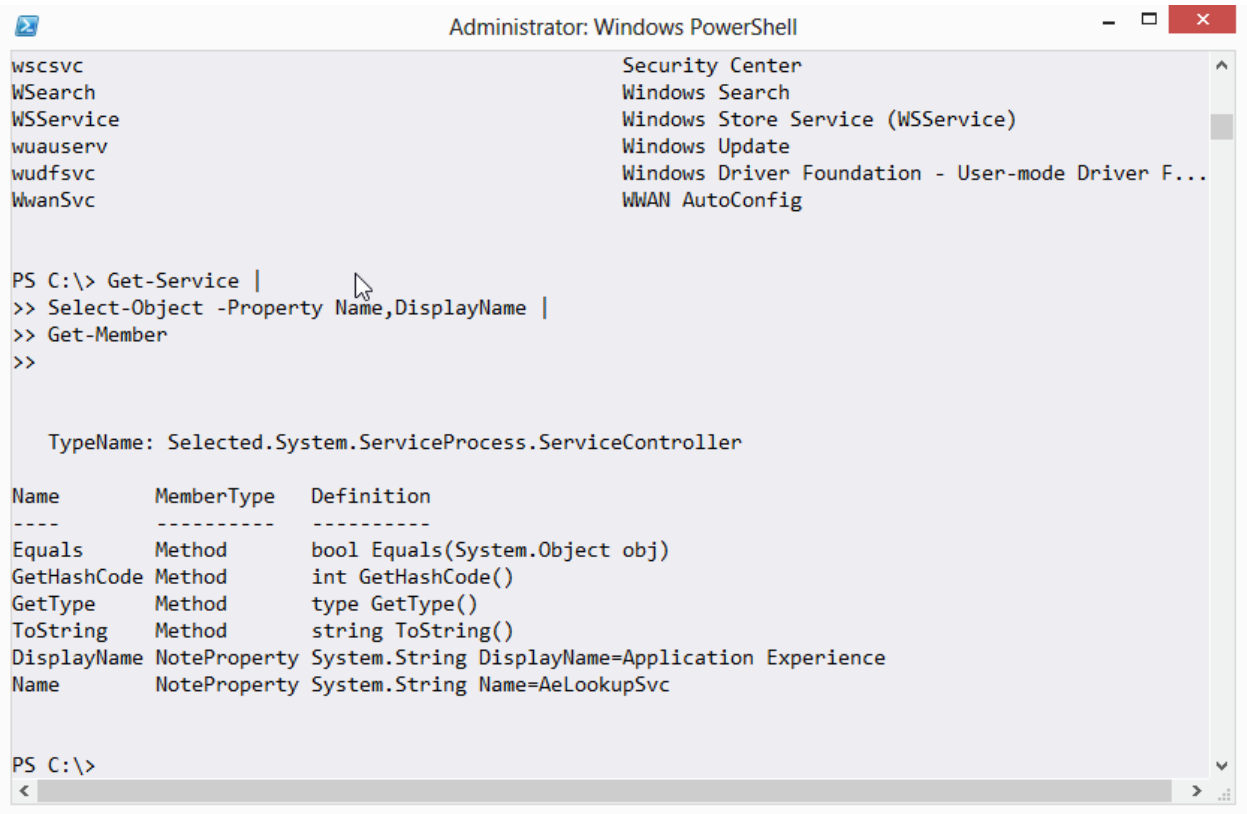



I mean, I'm pretty sure I have some running services, which is what this was supposed to display.

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.




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.




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

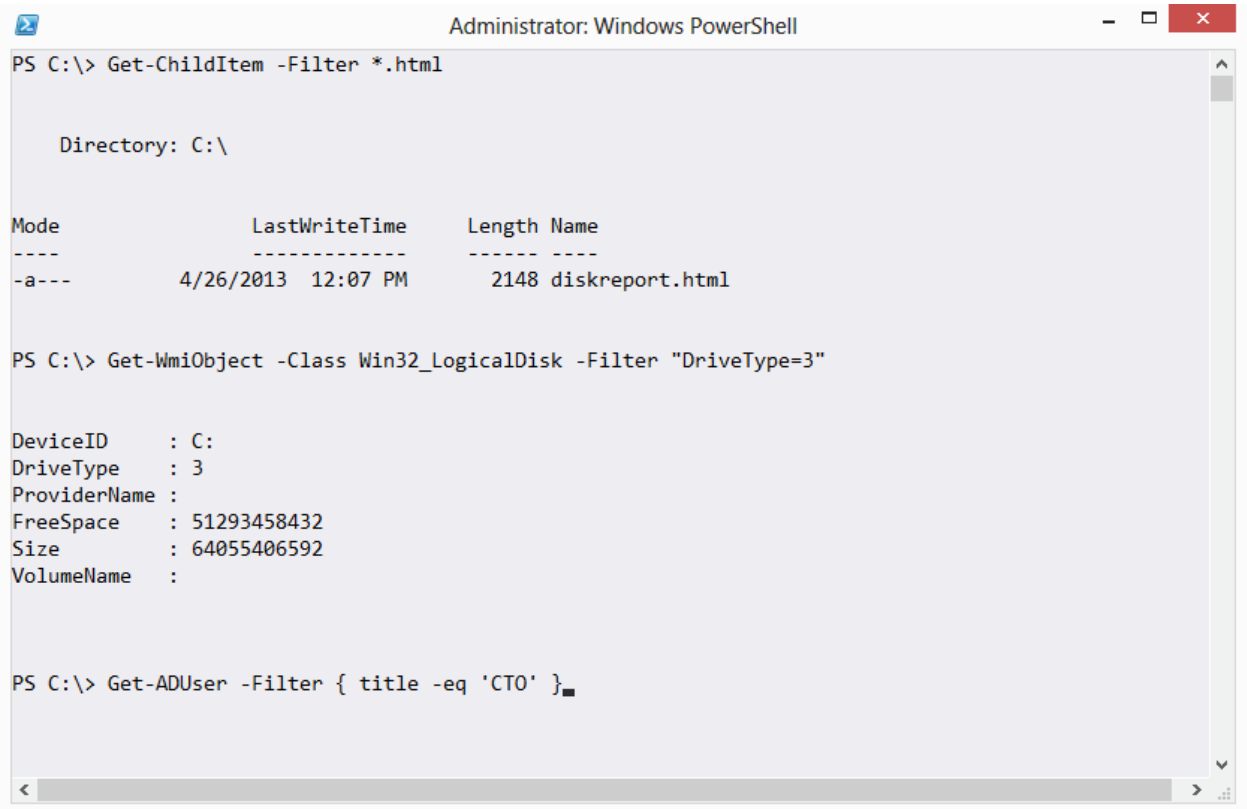
*And my Where-Object command was checking the Status property.* Do you see a Status property? No, you do not. My error is that I *removed* the Status property when I didn't include it in the property list of Select-Object. So Where-Object had nothing to work with, so it returned nothing.

(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:




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).

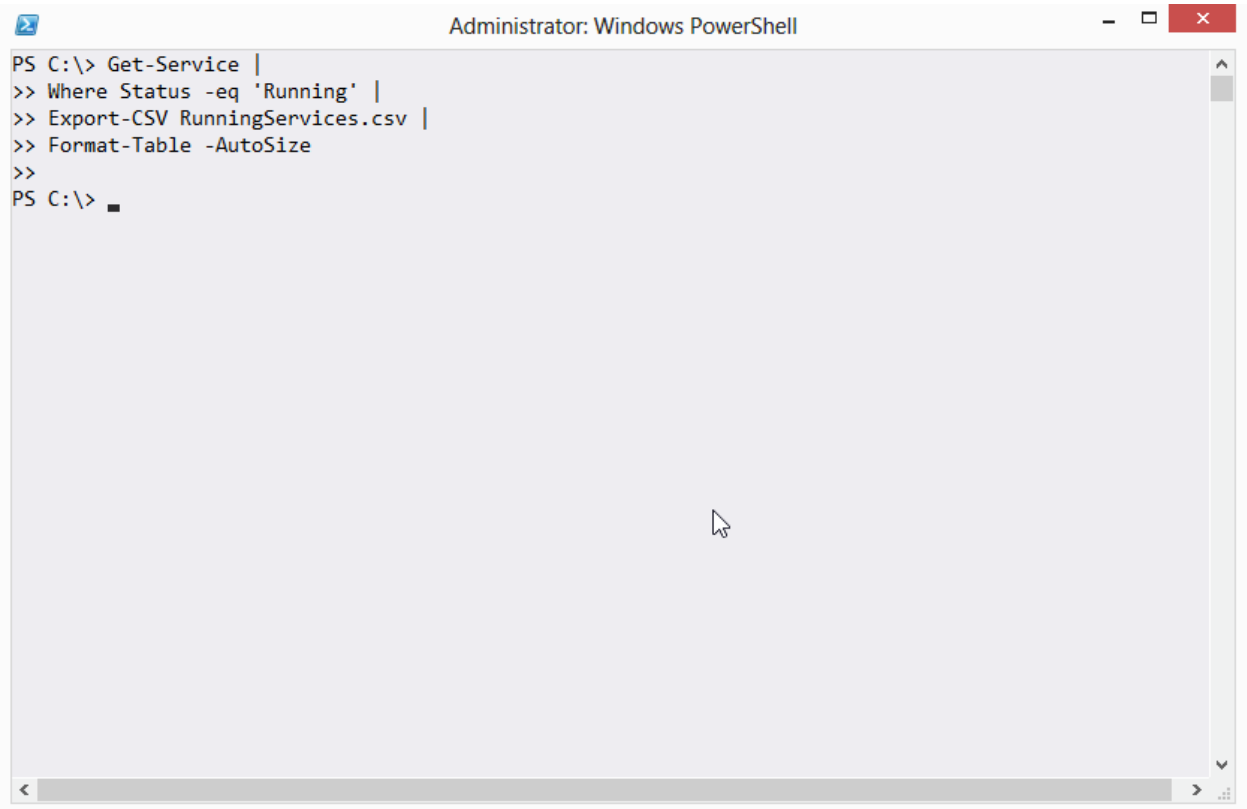
With Get-ADUser, -Filter wanted a script block, and accepted PowerShell-style comparison operators (like -eq for equality).

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.

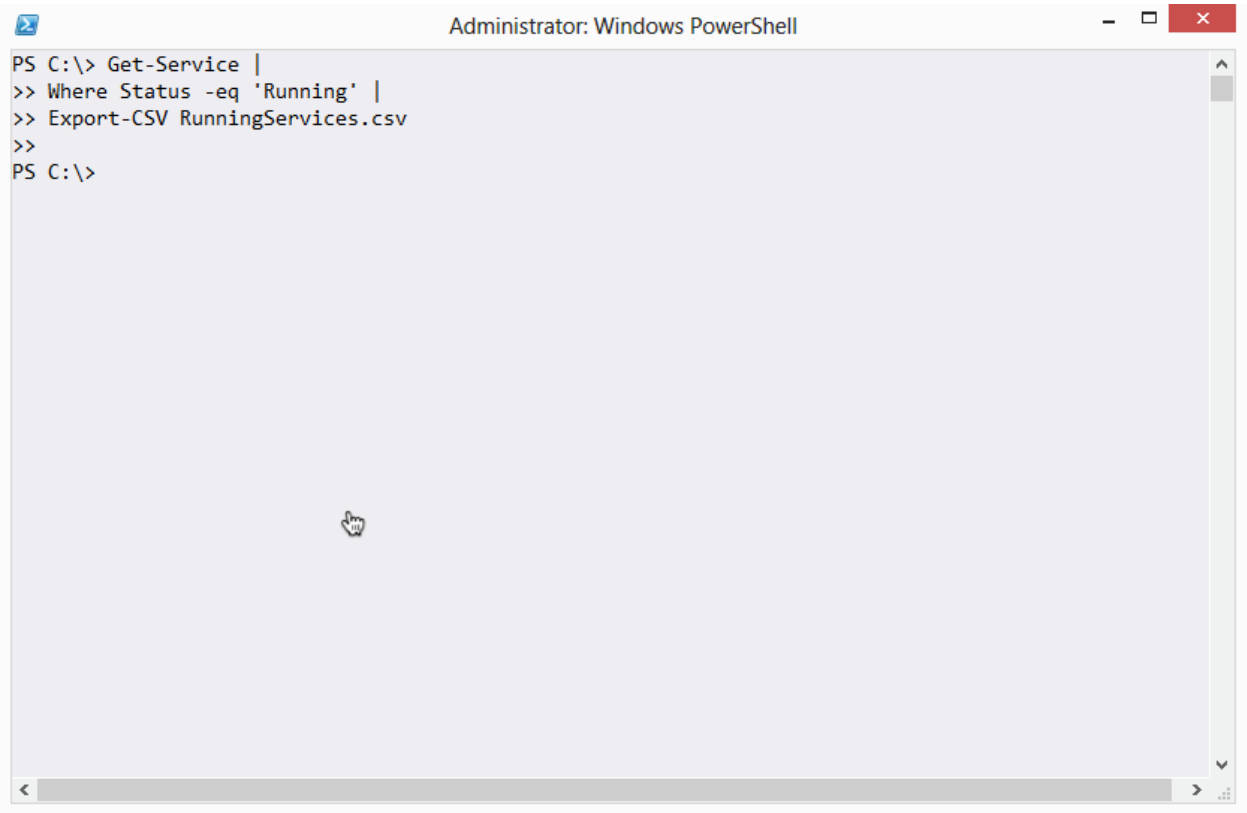
# Not Everything Produces Output

I see this one a lot in classes:




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:

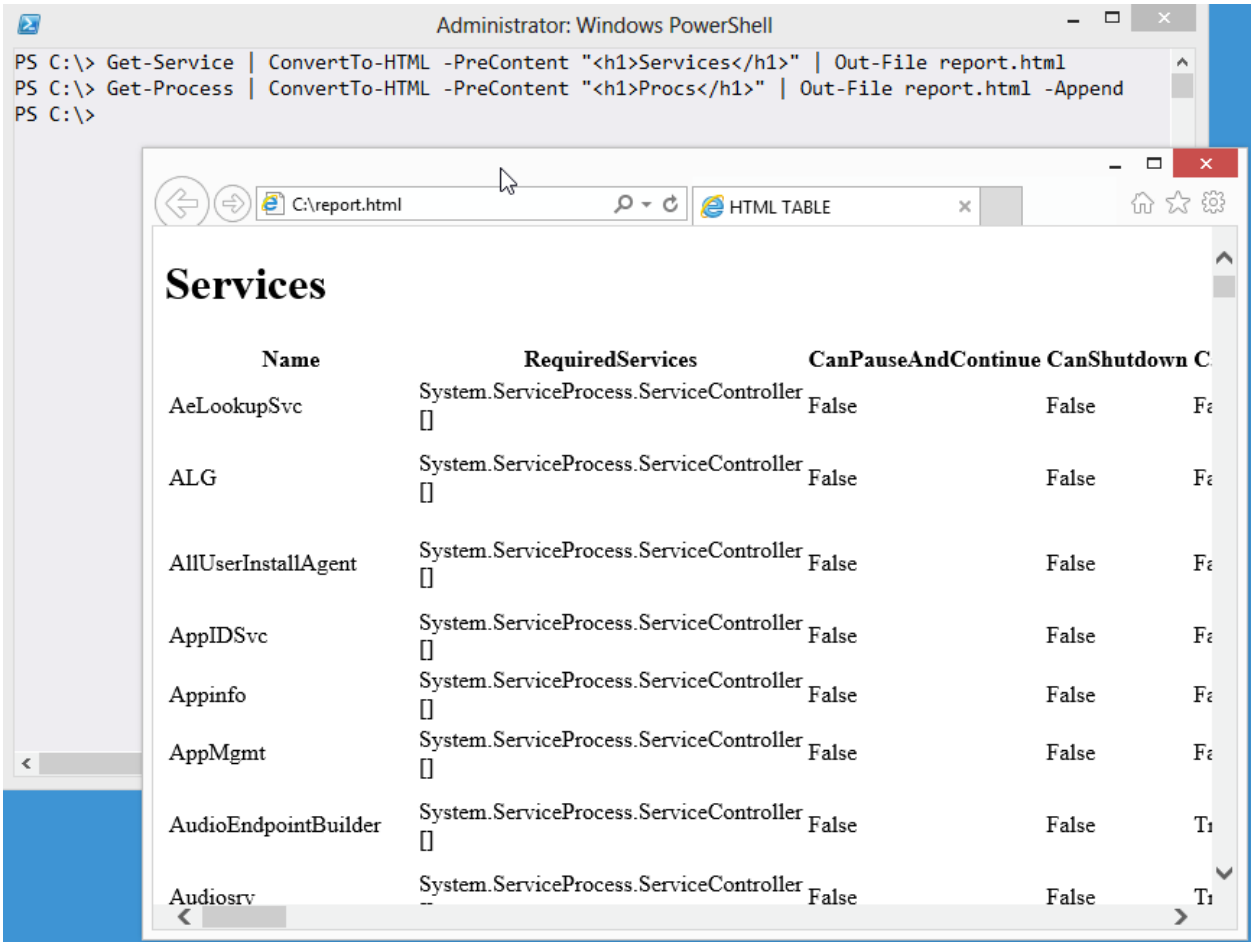



See? Nothing. Since there's nothing on the screen, there's nothing *in the pipeline*. You can't pipe Export-CSV to another command, *because there's nothing to pipe.*

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.

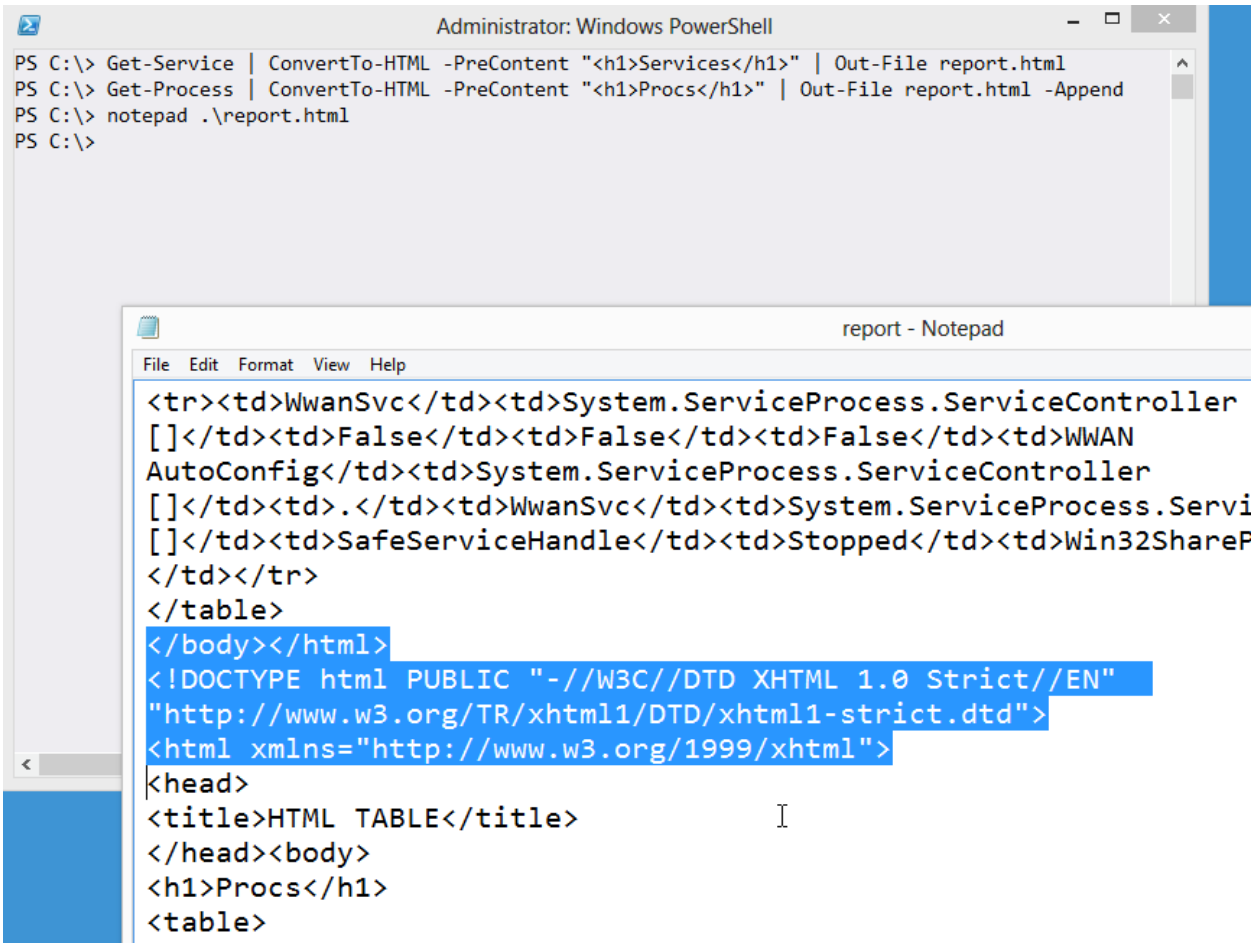
# One HTML Page at a Time, Please

This drives me batty:




What's happening is that someone ran two 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:

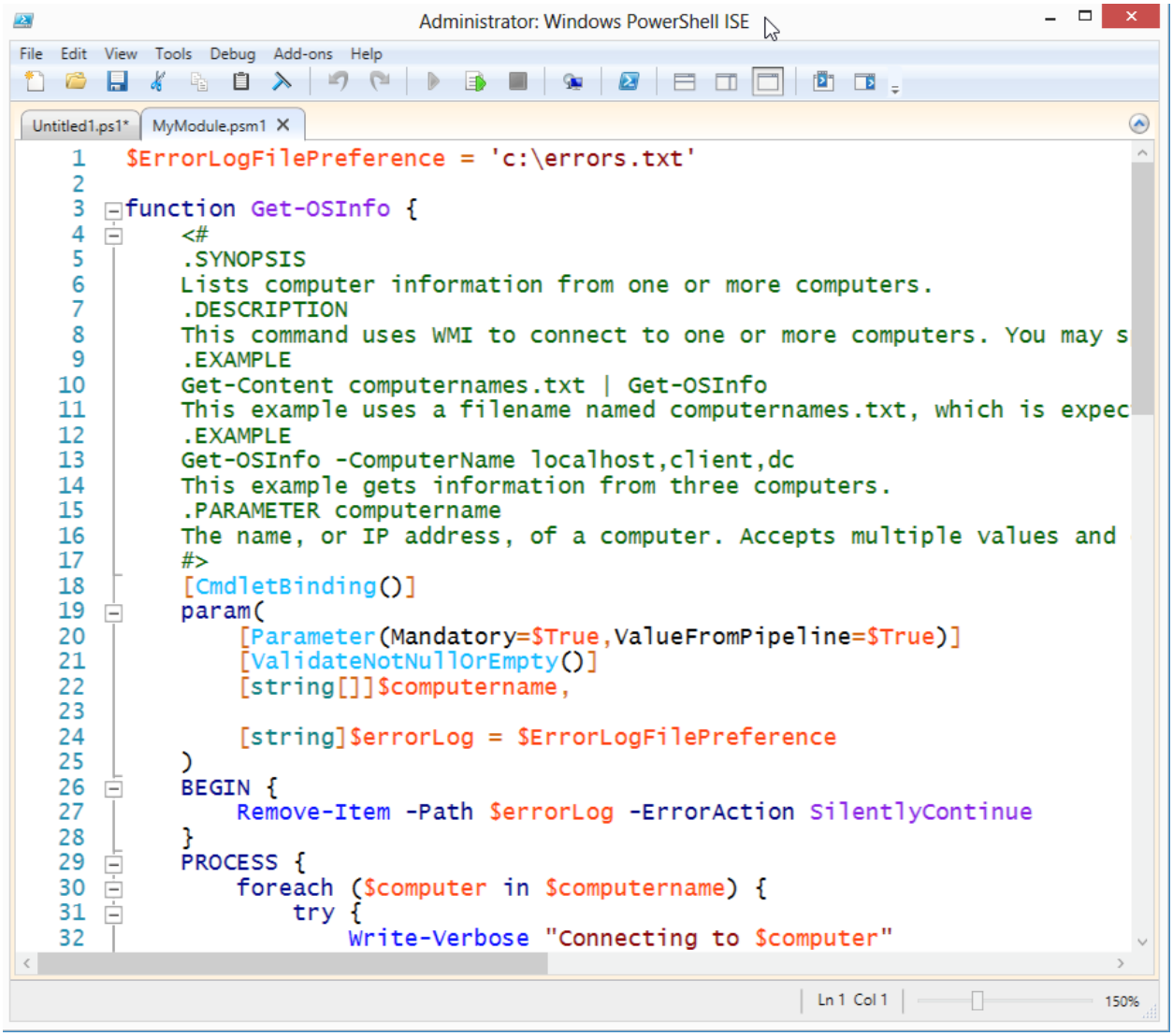



I've highlighted the lines that end one HTML page and start the next one. This is technically a malformed HTML file. It becomes tough to use this with some Web browsers (Firefox 20 is choking it down, but my current Webkit browsers aren't), tough to parse if you ever need to manipulate it programmatically,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




(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.

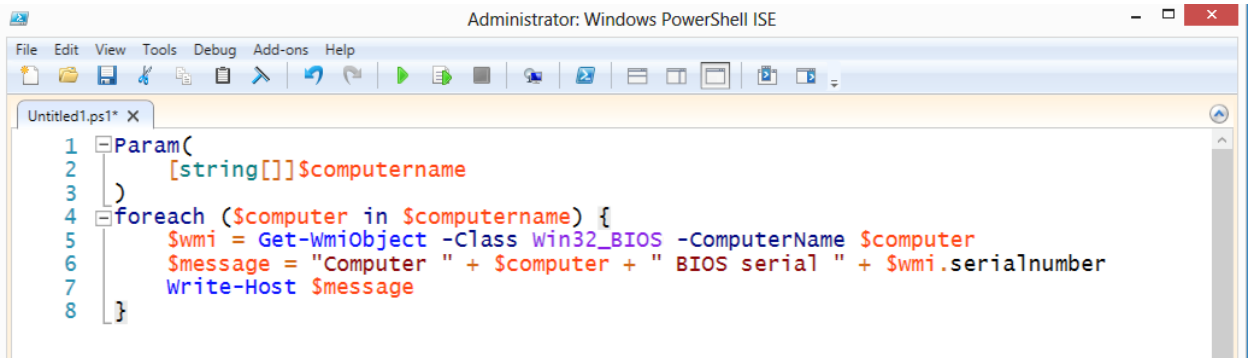
[Square brackets] are used around some attributes, like [CmdletBinding()], and around data types like [string], and to indicate arrays - as in [string[]]. They pop up a few other places, too.

{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 @{}).

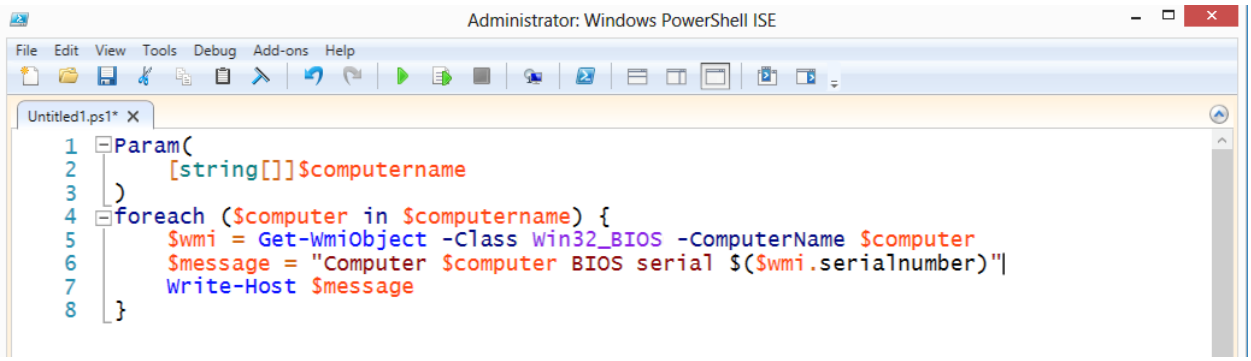
If your keyboard had a few dozen more buttons, PowerShell probably wouldn't have had to have all these overlapping uses of punctuation. But it does. At this point, they're pretty much just part of the shell's "cost of entry," and you'll have to get used to them.

# Don't Concatenate Strings

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




And completely unnecessary, when you use double quotes.



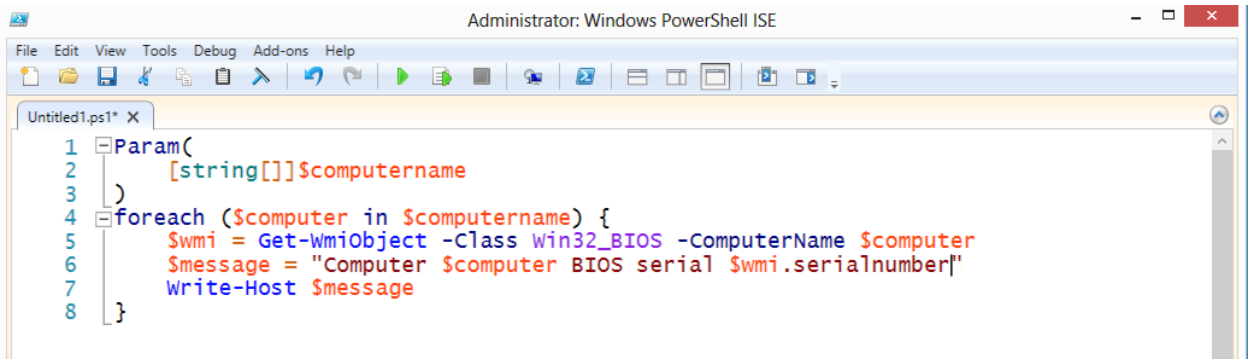

Same end effect. In double quotes, PowerShell will look for the $ character. When it finds it:

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**.

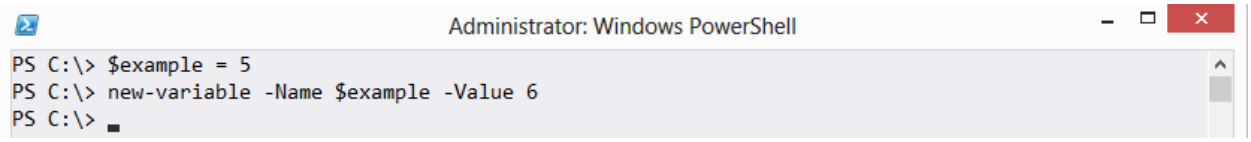
There's a sub-gotcha here:



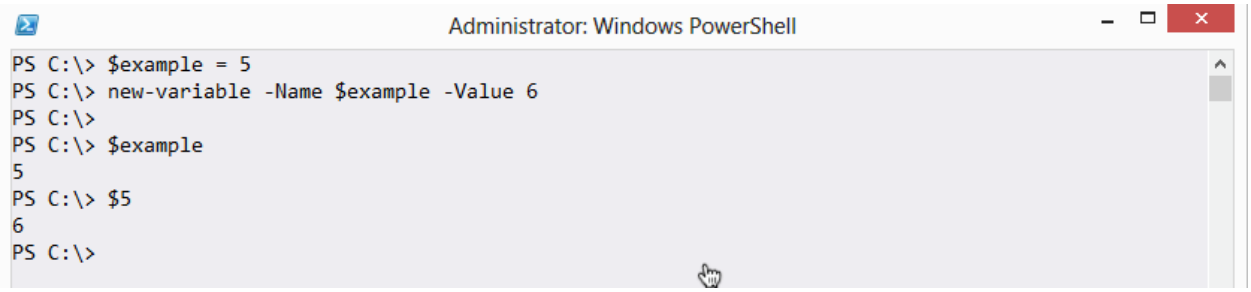

This won't work as expected. In most cases, $wmi will be replaced by an object type name, and **.serialnumber** will still be in there. That's because **.** isn't a legal variable name character, so PowerShell stops looking at the variable with the letter **i.** It replaces **$wmi** with its contents. 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.



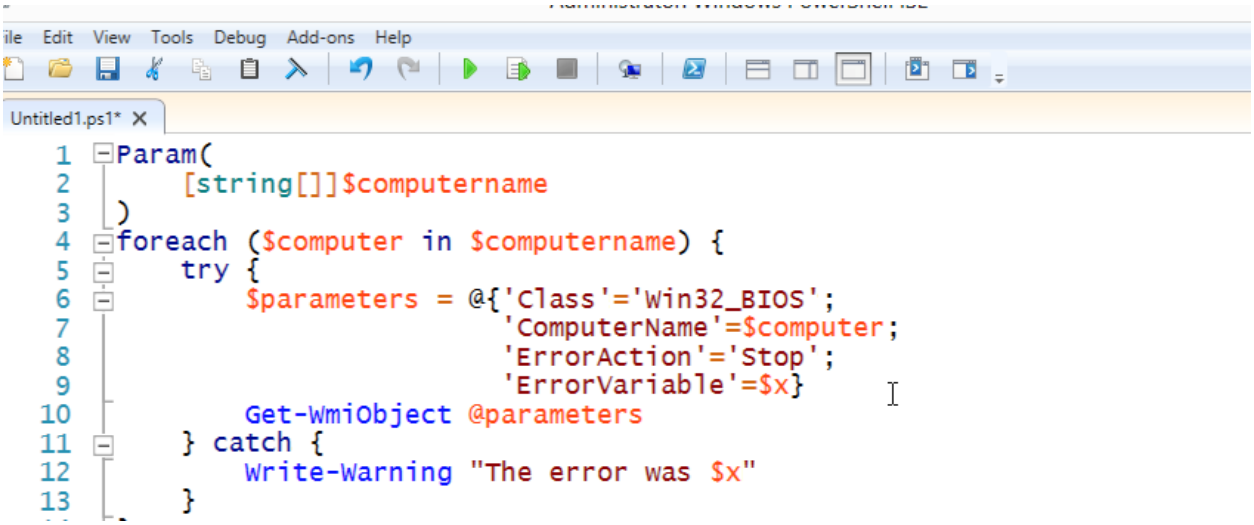

Can you predict what happened?



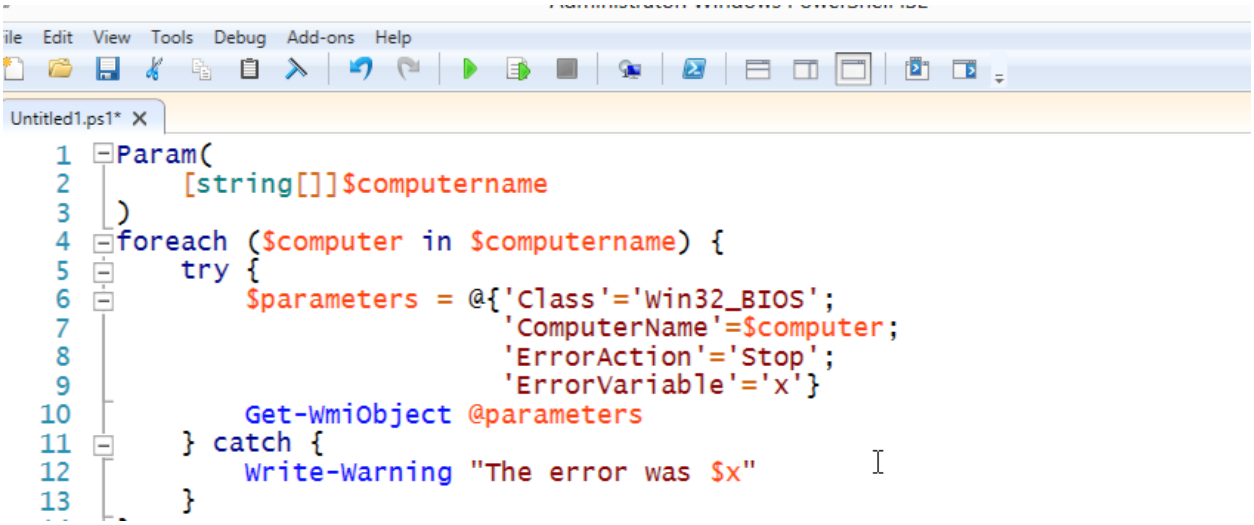

You see, the $ is not part of the variable's name. If you have a variable named **example**, that's like having a box with "example" written on the side. Referring to **example** means you're talking *about the box itself.* Referring to **$example** means you're messing with the *contents of the box.*

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:



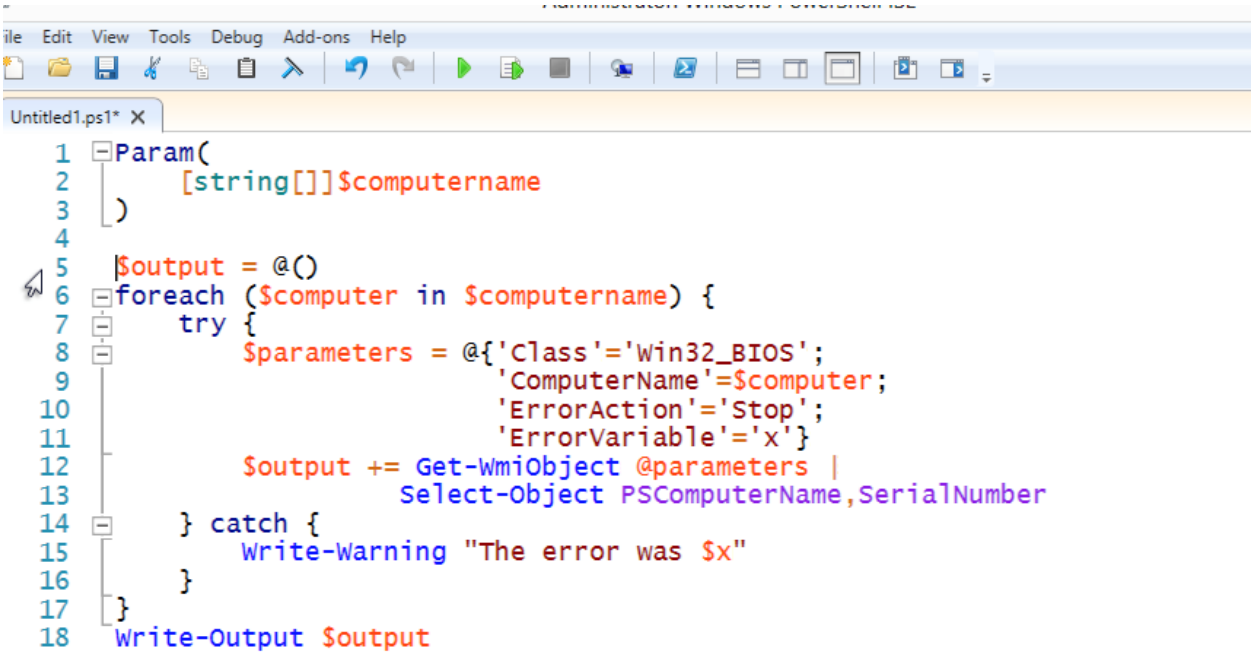

In that example, I used the -ErrorVariable parameter to specify a variable in which I would store any error that would occur. Problem is, I used **$x**. I should have used **x** by itself:




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.

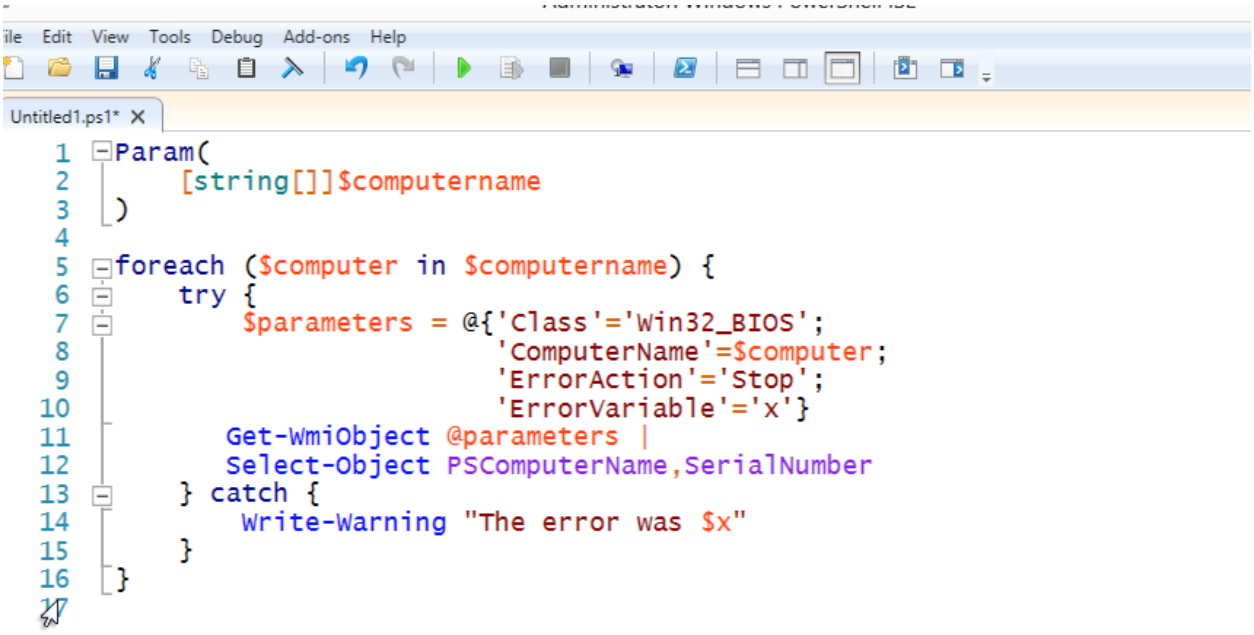
# Use the Pipeline, not an Array

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




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.

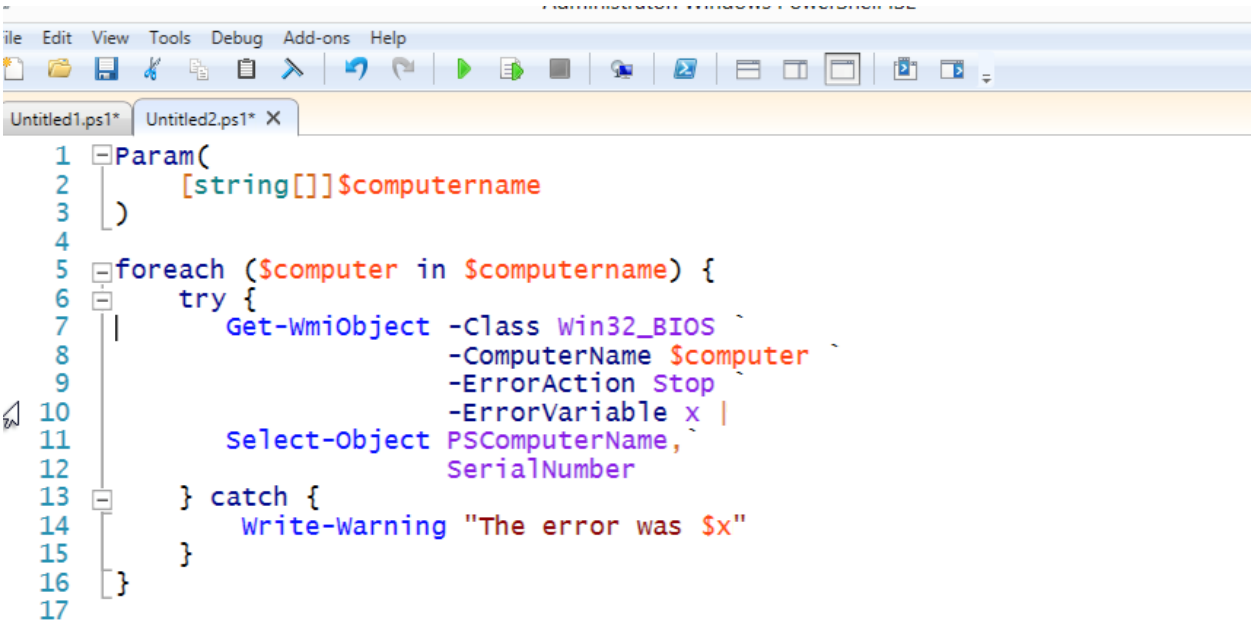
Poor practice. You see, this forces PowerShell to wait while this *entire* command completes. Any subsequent commands in the pipeline will sit their twiddling their thumbs. A better approach? Use the pipeline. Its whole *purpose* is to accumulate output for you - there's no need to accumulate it yourself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




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

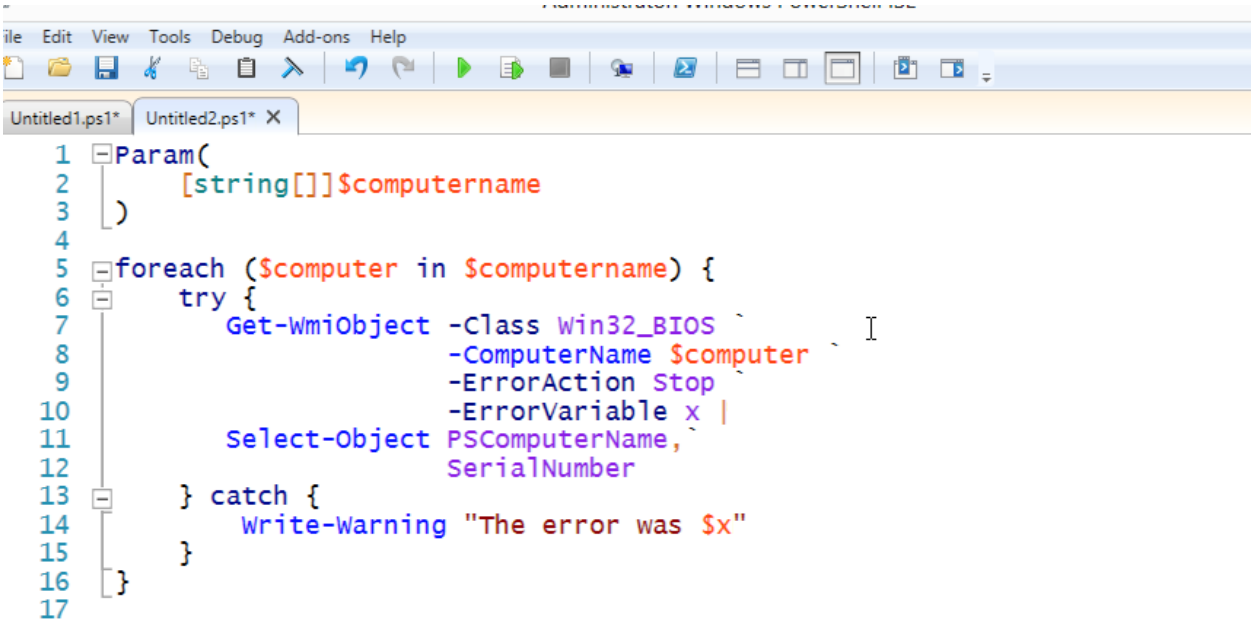
You'll see folks do this a lot:




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:

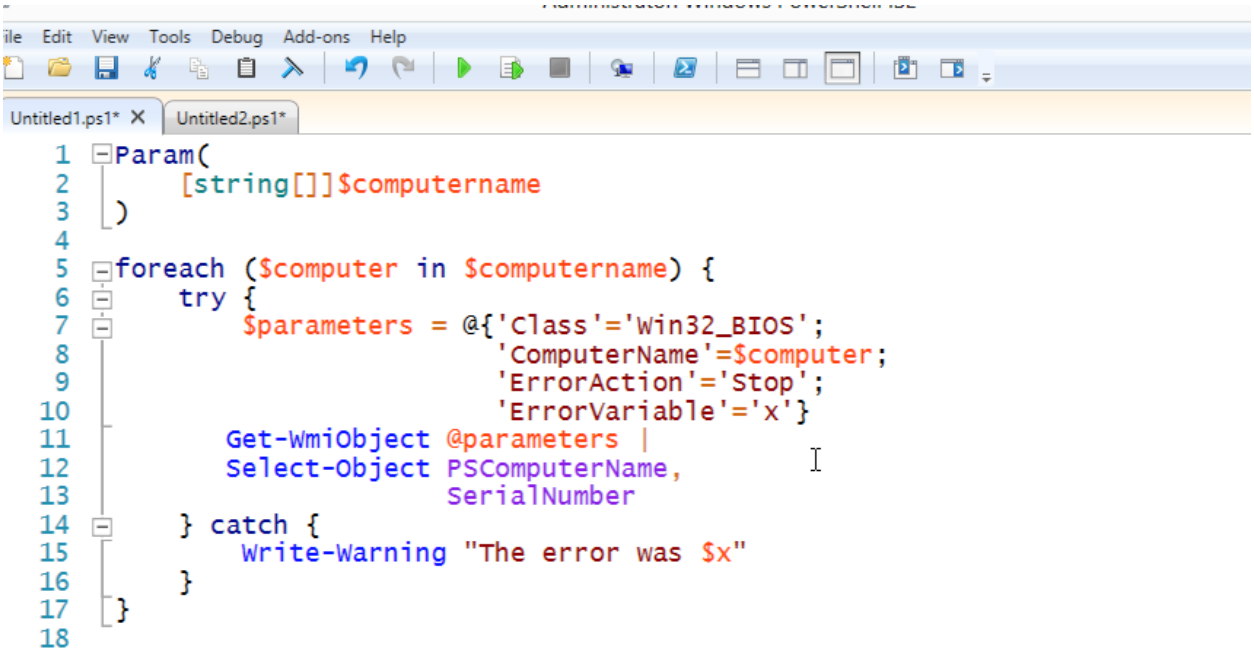



Carefully compare the -ComputerName parameter - in this second example, it's the wrong color for a parameter name, because I added a space after the backtick on the preceding line. IMPOSSIBLE to track these down.

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.




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.

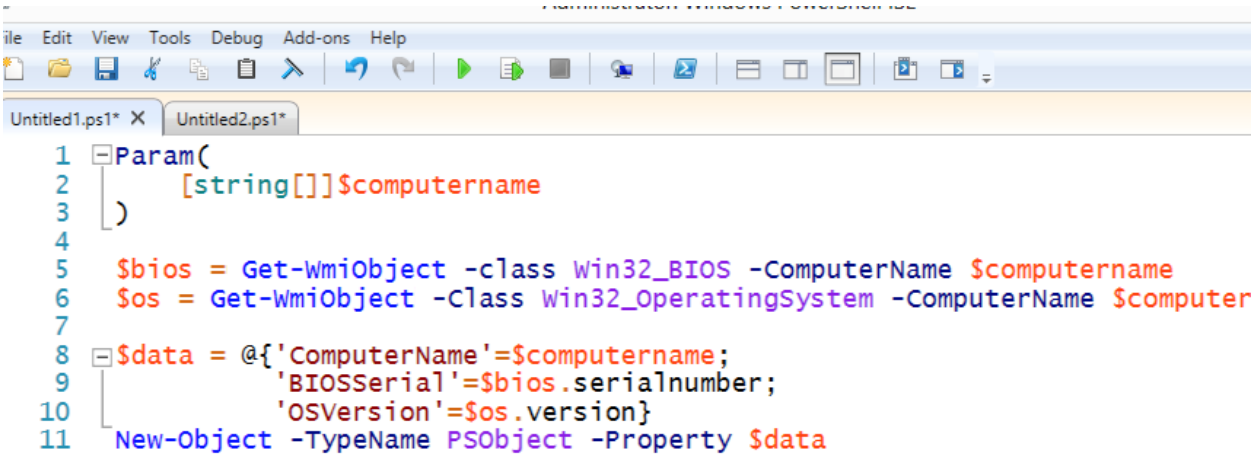
After Get-WmiObject, I have a pipe character - and you can legally line-break after that, too.

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

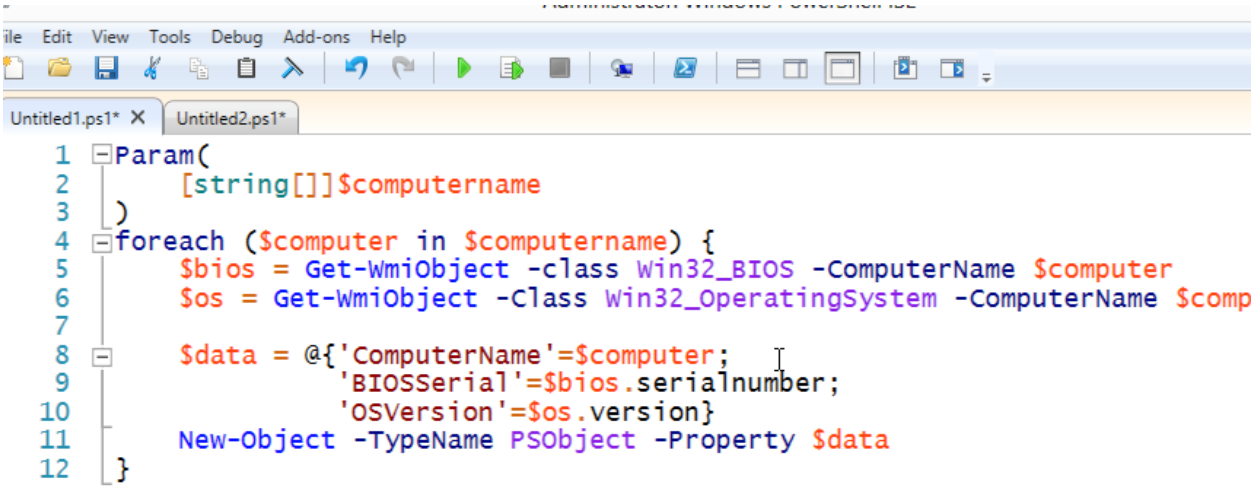
So I end up with formatting that looks at least as good, if not *better,* because it doesn't have that little ` floating all over the place.

# A Crowd isn't an Individual

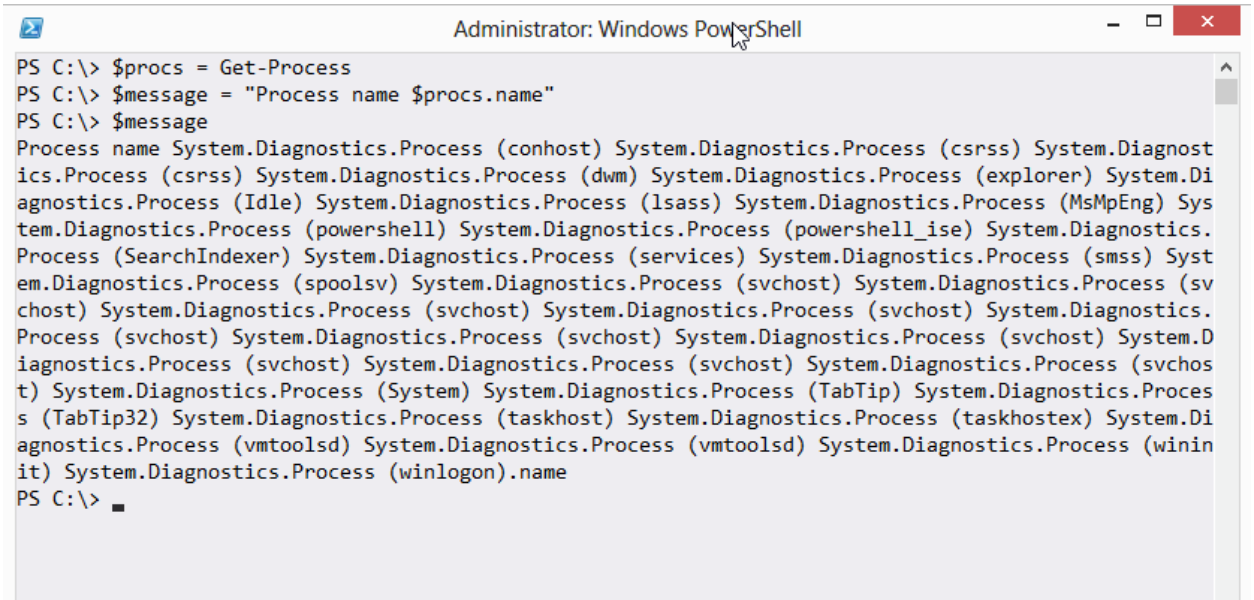
A very common newcomer mistake:




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:

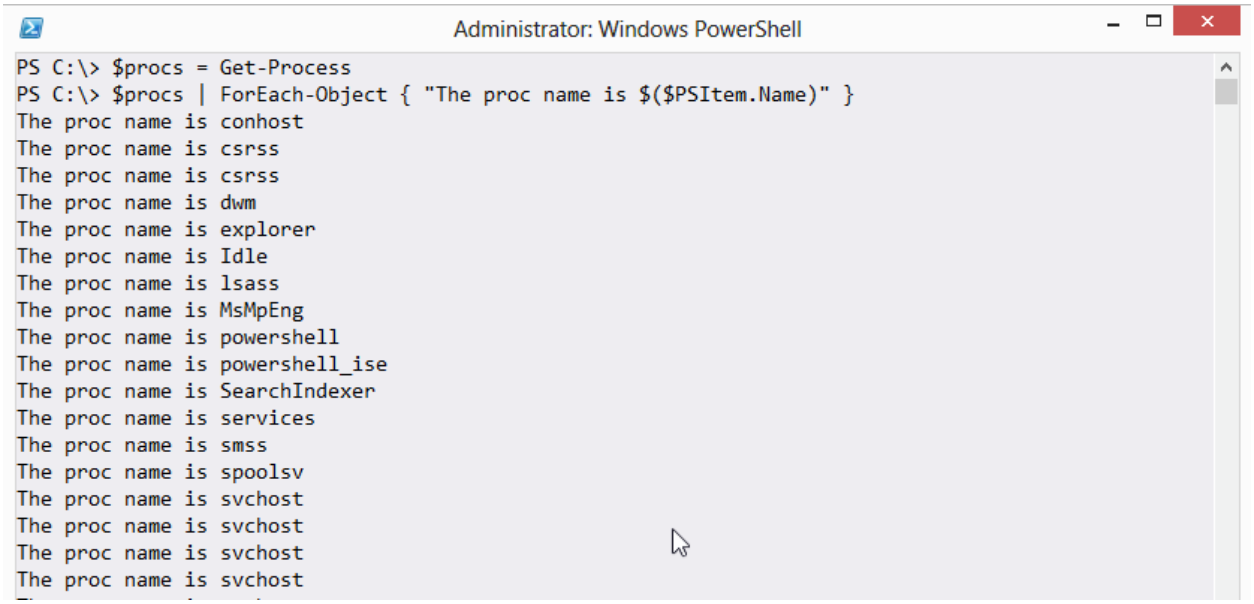



Folks will run into this even in simple situations. For example:



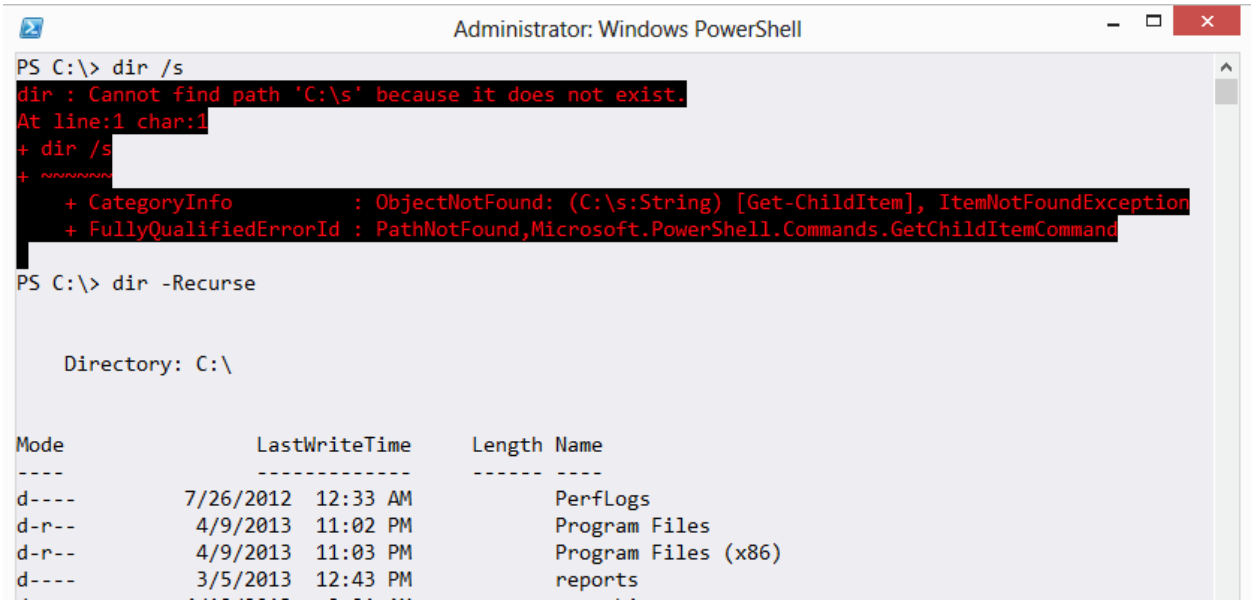

PowerShell v2 won't react so nicely; in v3, the variable inside double quotes is **$procs**, and since that variable contains multiple objects, PowerShell implicitly enumerates them and looks for a Name property. You'll notice ".name" from the original string appended to the end - PowerShell didn't do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

You'd probably want to enumerate these:



# 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.




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