The Administrator Crash Course

Windows PowerShell v2

Realtime publishers

Don Jones

Copyright Statement

PowerShell Crash Course Week 4

Welcome to the last week of our crash course! The previous 15 lessons have covered the majority of Windows PowerShell's main functionality. We'll continue building this week, covering intermediate techniques that may prove helpful from time to time.

Don't forget: I encourage you to continue exploring beyond this crash course, too. For example, visit http://windowsitpro.com/go/DonJonesPowerShell to find tips and tricks and FAQs and to ask questions, or drop by the PowerShell team's own blog at http://blogs.msdn.com/powershell for "insider" information. You'll also find a lot of in-person PowerShell instruction at events like TechMentor (http://www.techmentorevents.com) and Windows Connections (http://www.winconnections.com). If you want to get a few PowerShell tips per week, you're welcome to subscribe to my Twitter feed, @concentrateddon. I focus almost exclusively on PowerShell, and I'll pass along any tips or articles that I find especially useful. I've also put up a new resource to help you find all the *other* good PowerShell resources: http://shellhub.com. It's a hand-picked list of things guaranteed to be useful and educational.

How to Use this Crash Course

I suggest that you tackle a single crash course item each day. **Spend some time practicing whatever examples are provided and trying to complete tasks that make sense in your environment.** Don't be afraid to fail: Errors are how we learn. Just do it in a virtual environment (I recommend a virtualized domain controller running Windows Server 2008 R2) so that you don't upset the boss! Each "Day" in this crash course is designed to be reviewed in under an hour, so it's a perfect way to spend lunch for a few weeks. This book will be published in five-day increments, so each chapter corresponds to a single week of learning.

Week 4, Day 1: Error Handling

Errors are inevitable. Some of them, however, you can anticipate—things like a computer not being available on the network or a permission being denied. Don't listen to the Internets and just set $ErrorActionPreference = "SilentlyContinue" at the top of a script: That's a good way to suppress useful error messages that you *didn't* anticipate. Instead, locate commands that you think might cause an error, and add the parameter -EA "Stop" to them. Then, wrap the command in a Try-Catch block:

```
Try {
  Get-Service -computer $computername -ea Stop
} Catch {
  "Could not reach $computername" | Out-File log.txt -append
}
```

That's the smart way to deal with errors. In this example, I'm logging the failed computer name to a file (within double quotes, PowerShell will replace the $computername variable with its contents—cool trick, right?)

Week 4, Day 2: Debug Trace Messages

Sometimes you might have a complicated script that just isn't doing what you expect it to. 99% of the time, I find that those problems come directly from a property value, or a variable, that contains a value other than what you expected. For example, I once wrote a script like this:

```
$result = Test-Connection $computername
if ($result) {
 Get-Service -computername $computername
}
```

I was trying to ping the computer before attempting to connect to it, but this code never worked. So I added some debug code:

```
$DebugPreference = "Continue"
$result = Test-Connection $computername
Write-Debug $result
if ($result) {
 Write-Debug "Trying to connect"
 Get-Service -computername $computername
} else {
 Write-Debug "Not trying to connect"
}
```

Notice that I enabled debug output on the first line, and added an Else to my construct so that I'd get some kind of output either way. Write-Debug is neat in that you can set $DebugPreference = "SilentlyContinue" and it'll shut off the debug output without you having to go manually rip out all the Write-Debug commands you added.

This didn't solve my problem, but it did make me realize that Test-Connection wasn't returning a True or a False but rather a collection of objects. I read the help on the cmdlet (should have done that to start with, I guess), and found the change I needed to make:

```
$DebugPreference = "Continue"
$result = Test-Connection $computername -count 1 -quiet
Write-Debug "Ping is $result"
if ($result) {
 Write-Debug "Trying to connect"
 Get-Service -computername $computername
} else {
 Write-Debug "Not trying to connect"
}
```

Huzzah! My problem came from the fact that I was expecting $result to contain one thing, but it actually contained something quite different. Getting insight into the actual contents of the variable pointed me in the direction of my solution. That's what debugging is all about: getting inside your script's head.

Week 4, Day 3: Breakpoints

In a long script, you could easily wind up scanning through hundreds of lines of debug output. Yuck. An alternative is to just let the script run to the point where you think there's a problem, then let the script pause so that you can manually examine variables, properties, and whatnot. You can do this with a *breakpoint.*

You basically set a breakpoint for your script by giving your script's path and filename. Then you can decide if the script is going to break when you hit a particular line in the script, when a particular variable is read or written, or when a particular command is executed. It's all done with the Set-PSBreakpoint cmdlet:

```
Set-PSBreakpoint -script c:\mine.ps1 -line 37
```

Breakpoints take effect immediately, so go ahead and run your script. When you do, you'll get a very special prompt when the script hits the breakpoint. Within that prompt, you can examine variables by just entering the variable name. When you're done looking around, run Exit to let the script resume executing.

When you're finished with your breakpoints, you can delete them all:

```
Get-PSBreakpoint | Remove-PSBreakpoint
```

And you're done. This is a set of techniques that takes a wee bit of practice to get used to, but it's a powerful and useful technique once you do.

Note

A lot of third-party PowerShell tools provide their own breakpoint functionality, and they're often much enhanced over the basic capability. They'll list out variables for you, let you visually manage breakpoints, and so forth. I'll cover some tools in the last lesson.

Week 4, Day 4: WMI

I can't believe I let you go all the way to the last week, on the next-to-the-last day, to cover Windows Management Instrumentation! A WMI crash course is definitely in order here.

WMI is a technology that's been included in Windows since Windows 2000—and it's even an add-in feature for Windows NT 4.0, if you still have any of that lying around. Microsoft expands it a bit in each new version, and many products—like Office, SQL Server, and so forth—can all extend WMI as well. At its core, WMI is primarily about querying management information; there are some opportunities to make configuration *changes,* but they're a bit rare, and I'm not going to dive into them in this little crash course.

WMI is organized into *namespaces*, with the root\v2 namespace being the default. The namespaces contain *classes,* and those classes represent the manageable components that you can gather information from. An *instance* of a class actually represents an existing component. So, if you have two disks, then you have two instances of the Win32_LogicalDisk class. To query a class, you just need to know the computer it's on and the class name (and the namespace, if the class isn't in the default namespace). What you get back is an object, just like the objects produced by any PowerShell cmdlet. You can pipe them to Get-Member to see their properties, and those properties contain the management information you're after:

```
Get-WmiObject -class Win32_OperatingSystem -computer localhost
```

Technically, you don't need the -computer parameter if you want to query the local host. If you're querying a remote computer, you have the additional option of specifying a -credential parameter, where you can provide an Administrator username if the account you used to open PowerShell isn't an admin on the remote machine. The -computername parameter can even accept a comma-separated list of computer names, and it'll contact them sequentially (and skip over any ones that aren't available). All good tricks.

If you have the ActiveDirectory module that comes with Windows Server 2008 R2, a command like this will query every computer in the domain:

```
Get-WmiObject -class Win32_BIOS -computer ( Get-ADComputer -filter * | Select -
expand name )
```

Again, neat trick. Add a -searchBase parameter to Get-ADComputer to have it start in a specific OU rather than at the root of the domain.

WMI output can be formatted, exported, and converted, just like the output of every other command you've worked with. The trick with WMI is in finding the right class for what you want, and sadly there's no good way to do it. A WMI Explorer or Browser helps (I'll mention one in the next section), and Google is, of course, your friend. Google is also a good way to locate what WMI documentation is out there: Punch a WMI class name into your search engine and the first couple of hits will likely be to Microsoft's MSDN Library site, where the WMI documentation that does exist is kept. Don't expect to find documentation for every WMI class—Microsoft just hasn't been consistent about documenting them, unfortunately.

Over the long term, admins may find themselves dealing directly with WMI less and less, as Microsoft moves admin functionality into more consistent, better-documented cmdlets. Until then, WMI remains your best bet, in many cases, for retrieving management information.

Week 4, Day 5: Tools

I don't think anyone can be maximally-successful at Windows PowerShell without a few good supplementary tools. Here are some of my favorites:

- A PowerShell Help viewer, such as the free one at http://www.primaltools.com/downloads/communitytools/, makes it easier to browse and read help while typing commands in another window.
- A WMI Explorer (http://www.primaltools.com/downloads/communitytools/ again) makes it easier to find WMI classes and to see the properties and methods they offer.
- Want to build graphical dialog boxes into your PowerShell scripts? Consider PrimalForms (http://www.primaltools.com/products/), a commercial product that lets you drag-and-drop a GUI, and produces the script needed to make that GUI a reality.
- You need a better editor if you're going to get into scripting. Try these:
    - http://PrimalScript.com is a mature, feature-rich environment. A studio edition includes the GUI-builder tool I mentioned earlier.
    - http://PowerShellPlus.com is a PowerShell-only editor that incorporates a beefed-up command-line experience in addition to script editing.
    - http://PowerGUI.org offers a free editing experience that helps write scripts for you; a pro version (http://quest.com/powerguipro) includes options for running PowerShell scripts from your mobile device.

You'll find more on http://ShellHub.com. All of these commercial tools offer free trial editions, so there's no reason not to try them out and see which one suits you best.

Month Wrap-Up

We've covered a lot of ground: PowerShell's core commands and functionality, building customized table output, debugging, error handling, sorting, remote control, background jobs, and tons more—all in a very short span of time. The best way to make it all make sense is to *dig in there* and start trying some of these techniques. You'll probably get stuck at some point—*ask for help!* You can hop on http://connect.ConcentratedTech.com and register to ask me a question directly (when asked where you took a class with me, just put Realtime Publishers). I also encourage you to follow me on Twitter @concentrateddon. My tweets are almost exclusively about newly-posted PowerShell tips and content, articles and videos, and more. And, don't forget about all the other great *free* books on offer at http://nexus.realtimepublishers.com!

Download Additional Books from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this book to be informative, we encourage you to download more of our industry-leading technology books and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.