The Administrator Crash Course

Windows PowerShell v2

Realtime publishers

Don Jones

Introduction to Realtime Publishers

by Don Jones, Series Editor

For several years now, Realtime has produced dozens and dozens of high-quality books that just happen to be delivered in electronic format—at no cost to you, the reader. We've made this unique publishing model work through the generous support and cooperation of our sponsors, who agree to bear each book's production expenses for the benefit of our readers.

Although we've always offered our publications to you for free, don't think for a moment that quality is anything less than our top priority. My job is to make sure that our books are as good as—and in most cases better than—any printed book that would cost you $40 or more. Our electronic publishing model offers several advantages over printed books: You receive chapters literally as fast as our authors produce them (hence the "realtime" aspect of our model), and we can update chapters to reflect the latest changes in technology.

I want to point out that our books are by no means paid advertisements or white papers. We're an independent publishing company, and an important aspect of my job is to make sure that our authors are free to voice their expertise and opinions without reservation or restriction. We maintain complete editorial control of our publications, and I'm proud that we've produced so many quality books over the past years.

I want to extend an invitation to visit us at http://nexus.realtimepublishers.com, especially if you've received this publication from a friend or colleague. We have a wide variety of additional books on a range of topics, and you're sure to find something that's of interest to you—and it won't cost you a thing. We hope you'll continue to come to Realtime for your educational needs far into the future.

Until then, enjoy.

Don Jones

Copyright Statement

PowerShell Crash Course Week 1

By now you've probably gotten the message loud and clear that Windows PowerShell is pretty important; Microsoft is adding it to more and more products, and going forward, the company's plan is to incorporate PowerShell throughout all of its business products as a baseline administrative layer. No, the GUI isn't going away—it's still called "Windows," after all—but in most cases, the GUI will simply be running PowerShell commands under the hood. In some cases, the GUI will be "de-emphasized," meaning the GUI might not surface *all* of the product's administrative functionality.

If you're ready to get started in PowerShell, and have no experience, this is the crash course for you. If you have a bit of Unix or VBScript experience, try to remove that from your brain: PowerShell will look familiar, but it's really something very new and different.

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).

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.

By the way, this crash course isn't intended to be comprehensive—I already co-authored *Windows PowerShell v2: TFM* and don't intend to re-write the same book here! Instead, this is designed to get you up and running quickly with the *most crucial elements* of the shell. I'm skipping over a lot of stuff to get right to the really good bits, and in some cases, I may gloss over technical details simply because they don't contribute to speedy understanding of the most important things. You should obviously keep exploring; in my blog (linked earlier), for example, I go into a lot of these little details in one short article at a time. You'll also find PowerShell video tips on http://nexus.realtimepublishers.com, and those can help you embrace some of the more-detailed things that I might skip here.

Pre-Requisites

PowerShell v2 comes preinstalled on Windows 7 and Windows Server 2008 R2; it's available as a free download (from http://download.microsoft.com) for Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008. Be sure to download the right version for your operating system (OS) and architecture (32- or 64-bit). You'll need, at a minimum, .NET Framework v2 installed. Ideally, get the latest version of the Framework, or at least v3.5 Service Pack 1, because that enables maximum Windows PowerShell features.

Windows Server 2008 R2 ships with a number of PowerShell modules that connect PowerShell to Active Directory (AD), IIS, BitLocker, and other technologies. Generally speaking, these modules will only run on Windows Server 2008 R2 or, if you install the Remote Server Administration Toolkit, on Windows 7. There's a trick called *implicit remoting* (I'll get to it) that lets you access these cmdlets from older OSs that have PowerShell v2 installed, but you'll need at least one Win7 or Win2008R2 machine on your network to *host* the cmdlets for remote use.

Week 1, Day 1: Commands, Cmdlets, and Aliases

PowerShell is *not,* first and foremost, a scripting language. It's a shell, not unlike Cmd.exe. It's written in .NET instead of C++ or something, but in the end it's a text-based command-line window. You type commands, hit Enter, and they run—and you see results.

Go on, try it. Run **Ping**, **Ipconfig**, **Net Share**, or whatever other commands you may know. They'll work. Those three specifically are *external commands,* meaning they exist as standalone .exe files. PowerShell also has native commands, which are called *cmdlets* (pronounced "command-lets"). The difference with these is that they *only* run within PowerShell; you can't get to them from Cmd.exe or from Explorer or anywhere else. Examples include **Dir**, **Cd**, **Del**, **Move**, **Copy**, and **Type**. Yep, those look just like command names you're probably familiar with. You can try **Ls**, **Cat**, and **Cp** while you're at it, because those will all work, too.

But they won't work in quite the same way that you're used to. Those are actually *aliases,* or nicknames, to PowerShell cmdlets. The real cmdlet names are things like **Get-ChildItem, Set-Location, Remove-Item, Move-Item, Copy-Item,** and **Get-Content.** The aliases exist to give you something a bit easier to type that corresponds to the MS-DOS-style command names that you're probably familiar with. These new cmdlets—and their aliases—work similarly to those old-school commands, but they get there in a different way. For example, try running **Dir /s** in PowerShell; you'll probably get an error. Run **Help Dir** and you'll see why you got an error: There's no /s parameter.

In PowerShell, parameters all begin with a dash (-) not a slash (/), and parameter names tend to be full words, like *recurse.* So running **Dir -recurse** will work just fine. Actually, you don't have to type the full parameter name; you only need enough so that the shell can uniquely identify it. **Dir -r** will probably work fine. And no, you can't build an alias that would make "Dir /s" work; aliases are just a *nickname* for the cmdlet name. Aliases don't have any effect on the parameters of the cmdlet.

That **Help** command is going to be your new best friend—or it had better be, if you plan to master PowerShell. Run **Help *** for a list of all help topics; notice the numerous "about" topics that don't relate to a specific cmdlet but instead cover background concepts. That's your manual (in fact, you can use the Unix-style alias **Man** instead of **Help** if you prefer). Run **Help *service*** to see everything that has to do with services, or **Help *user*** to see if there's anything in there to deal with user accounts.

While we're at it, notice the cmdlet names: They have a specific naming pattern, consisting of a verb, a dash, and a *singular* noun (it's **Get-Service**, not "Get-Services"). The verbs come from a strictly-controlled set: It will always be **New** and not **Create,** and you'll never see **Delete** in place of **Remove**. These consistent verb names, combined with common nouns such as Service, Process, EventLog, and so forth, make it easier to guess at a cmdlet name. Can you guess the cmdlet provided by Exchange Server to retrieve mailboxes? **Get-Mailbox.** What cmdlet in the AD module might retrieve a user account? **Get-ADUser**. Yeah, sometimes you'll see a product-specific prefix, such as "AD," attached to the noun. That helps distinguish it from other kinds of user accounts that might exist in your environment.

Try running some single commands. Stuck for ideas? Run **Help *** to get a list of help topics, which will include all the cmdlets, then run **Help *cmdlet-name* -example**. Adding the **-example** parameter retrieves a list of examples for that particular cmdlet—very handy!

Week 1, Day 2: Output

Running **Dir** and so forth certainly produces text on the screen. But let's look at why text is bad.

In the Unix world, everything is text based. Run a command that generates a list of running processes, and you'll see a text list laid out as a columnar table. A Unix admin looking for a particular process name might pipe that text to a command like Grep, telling it to filter out entire rows based on the contents of columns 8 through 16, which contained process names. That kind of text parsing has been a common task in most shell-based administration. It's tough because it requires exactitude—and often a lot of experimentation—that goes beyond the actual administrative task. PowerShell doesn't work that way.

Instead, PowerShell cmdlets produce *objects,* which are essentially a specialized data structure in memory. Instead of putting information into tables and lists, information goes into this specialized structure. The benefit of this structure is that you can ask the shell for a single piece of information, and it can instantly retrieve it without you having to know exactly how the structure is built. In other words, you don't ask the shell to look at the contents of columns 8 through 16; you just ask it to look at the process' names. The shell knows which bit of the data structure contains the name so that you don't *have* to know.

This turns out to be pretty powerful, as you'll see shortly. But you might ask yourself why PowerShell cmdlets *still seem to produce text.* Well, that's because the shell knows that you, poor human being that you are, can't comprehend the wondrous data structures stored in your computer's memory. So when the shell finishes running commands, it converts all those objects into text-based tables and lists for you. Essentially, it retrieves *some* of the objects' attributes from the in-memory data structure and dynamically constructs a text-based table or list.

You can actually exercise a tremendous amount of control over this process, or just sit back and let the defaults take over. If you aren't liking the defaults, there are really three steps you need to take.

Step 1: Find Attributes

Every object that the shell works with has numerous attributes or *properties.* For example, a process has properties for its name, ID, memory consumption, and so forth. To see a list of them all, *pipe* the object to **Get-Member** (which has an alias, **Gm**). For example, **Get-Process | Gm** will show you the properties of a process; **Get-Service | Gm** will show you the properties of a service.

Hint

Any cmdlet that uses **Get** as its verb will usually produce some kind of object that can be piped to **Get-Member**.

Make a note of the properties that you think you'd like to see. Just write down their names, for now.

Step 2: Pick a Layout

PowerShell offers three default layouts: tables, lists, and a wide list. Decide which one you think will work for your needs. Keep in mind that tables can only hold so many columns without truncating information, so if you've selected a LOT of properties, a list might be appropriate. A wide list only displays a *single* property—something else to keep in mind. Once you've chosen a layout, you'll pipe your objects to it: **Get-Process | Format-List**, for example, or **Get-Service | Format-Wide**, or **Get-EventLog Security -newest 20 | Format-Table**. The aliases for those Format cmdlets are **FL, FW,** and **FT**, respectively.

Hint

The shell isn't case-sensitive about cmdlet or alias names. FT is the same as Ft, ft, and fT.

Step 3: Add Your Properties

By default, the shell decides what properties are shown in a table or list, and it defaults to the Name property for wide lists. Customize that by just providing a comma-separated list of properties: **Get-Process | FL Name,ID,VM,CPU**. If you just want every property shown, use * for the property list: **Get-Service | FL ***.

Additional parameters of the Format cmdlets enable further customization. See if you can answer these questions:

- If you use a table with only a couple of properties, such as **Get-Service | FT Name,Status**, you'll notice a lot of unused space on the screen. What parameter of **Format-Table** might eliminate that extra space and instead automatically size each column for its contents?
- If you include too many columns, **FT** may truncate their contents. What parameter would instead force it to word-wrap that information?
- **FW** defaults to two columns. How can you have a list of four columns?

Format, Then You're Done

A trick about the Format cmdlets is that they *consume* whatever you pipe into them. They output a special kind of formatting instruction that really only makes sense to the shell itself. Try running **Get-Process | FT | GM** and you'll see what I mean. The practical upshot of this is that a Format cmdlet will almost always be the *last thing* on the command line. "Almost always?" Yes—the one type of cmdlet that can understand Format output is an Out cmdlet: **Out-Host, Out-Printer, Out-File,** and so forth. In fact, **Out-Host** is the one used by default in the console window, but you can pipe formatted output to the other Out cmdlets to redirect output to a printer, a file, or elsewhere:

- If you direct output to **Out-File**, the file width defaults to 80 characters. What if you wanted to pipe out a much wider table of information? Is there a parameter that would let you modify the logical width of the file?
- What parameters allow you to specify a destination printer when piping output to a printer?
- Can you pipe output directly to an Out cmdlet without using a Format cmdlet, such as **Get-Service | Out-File test.txt** ?

Week 1, Day 3: The Pipeline

You've already started piping stuff from one cmdlet to another, so it should come as no surprise that PowerShell cmdlets run in a *pipeline.* Essentially, a pipeline is just a sequence of cmdlets:

Get-Service | Sort Status -descending | Format-Table -groupBy Status

The pipe | character separates each cmdlet. Each cmdlet places things into the pipeline, and they are carried to the next cmdlet, which does something with them. Then that cmdlet places something into the pipeline, which carries it to the next cmdlet…and so on. This can create some pretty powerful one-line commands, and thanks to the shell's cmdlet naming syntax, they can be pretty easy to figure out. For example, consider this pseudo-command:

Get-Mailbox | Sort Size -descending | Select -first 10 | Move-Mailbox Server2

That's not the exact correct syntax, but hopefully it conveys the power of the shell's cmdlet interaction. The trick is that every cmdlet gets to decide what kind of input it will accept. In other words, you can't just pipe anything to anything. This would make no sense:

Get-ADUser -filter * | Stop-Service

There's no reason why piping a user account to **Stop-Service** should make sense, and in fact it won't work. So how can you tell what a cmdlet is willing to accept as input from the pipeline? There are (as will become a theme in this crash course) three steps.

Step 1: Determine Your Output

First, cmdlets that produce output are, as we already learned, producing *objects*. The thing is, not all objects are built the same. A process looks very different from a service, for example, which is entirely different than an event log entry or a user account. So, each object has a *type name*, which simply describes the kind of object you're looking at. Piping objects to **Get-Member** reveals their type name. Run **Get-Service | Gm** and see if you can find the type name. Go on, I'll wait.

…waits…

It's a ServiceController, right? You can often just take the last segment of the type name. Now there's just one trick: All objects are technically of whatever type they are *and* they are the more generic "object" type. That's like saying you're a *Homo sapien*, which is a very specific type name, and that you're also an *organism*, which is much more generic. "Object" is just a very generic classification for object types.

Ok, so now you know what you have as your cmdlet output: A generic "object" as well as some more specific type name. Now, what can you do with it?

Step 2: Find Matching Input Types

Run **Help Stop-Service -full** (you'll find that the -full help is often the most useful). Start looking at the breakdown for each parameter. Notice how each parameter has the option to "Accept pipeline input?" It's False for many of them. In fact, for **Stop-Service**, the first one that's True is the -inputObject parameter. More specifically, it accepts pipeline input ByValue, it says in the help. Looking at the parameter definition, you'll see that the *type* of object it accepts is ServiceController. Wait, where have we seen that before?

So here's how it works:

1. You pipe object(s) from one cmdlet to another.
2. The receiving cmdlet looks at the *type name* of the incoming objects.
3. The receiving cmdlet looks to see whether any of its parameters will accept pipeline input ByValue for *that* type name.
4. If it finds one (and there will be zero or one, but not more), the input objects are "given" to that parameter.

So that's why this works (and it'll crash your machine, so don't run it):

Get-Service | Stop-Service

It works because **Get-Service** produced ServiceController objects. Those were piped to **Stop-Service,** which quickly realized that the -inputObject parameter was willing to accept objects *of that type* (which is what ByValue means). So those services were handed off to the -inputObject parameter, specifying the services that should be stopped.

Reading that help file a bit more, you'll notice that -Name also accepts input ByValue. Its value type is String, meaning if you pipe in a string of characters, they'll be attached to the -Name parameter, specifying the service(s) to stop:

"BITS" | Stop-Service

"BITS","TrustedInstaller" | Stop-Service

Tip
When you make a comma-separated list of values, PowerShell treats them as a single group, so those two values are piped in as a unit. Because they're both of the String type, they'll both attach to the -Name parameter, and both services will be stopped.

And how did we know that "BITS","TrustedInstaller" were strings? By using **Get-Member**, of course!

"BITS","TrustedInstaller" | GM

They're clearly identified as a System.String (just "String" for short) by the output of **Get-Member**.

Step 3: When Types Aren't Enough

Now, the shell isn't super-smart. It will *try* to do stuff that doesn't make sense, if *you* tell it to. For example, consider this:

Get-Process | Stop-Service

Makes no sense, right? Well, let's look at it from the shell's point of view. **Get-Process** produces objects that, according to **Get-Member**, are of the System.Diagnostics.Process type. Great. Looking through the help for **Stop-Service**, I don't see any parameters that will bind a Process object ByValue; I also don't see any that would bind the more-generic "object" ByValue. So accepting pipeline input ByValue will fail.

But the shell has a backup plan: Accepting pipeline input ByPropertyName. For **Stop-Service**, you'll see this only on the -Name parameter. What does this mean?

1. You pipe object(s) from one cmdlet to another.
2. The receiving cmdlet looks at the *type name* of the incoming objects.
3. The receiving cmdlet looks to see whether any of its parameters will accept pipeline input ByValue for *that* type name.
4. If it doesn't find one, which is the case in this example, it will look to see what parameters accept pipeline input ByPropertyName.
5. It will then attach those parameters to *the properties of the incoming object(s) that have a matching name.* In other words, if the incoming objects have a Name property, the value of that property will go into the -Name parameter of **Stop-Service** simply because the names match.

Remember, this is "Plan B," so it only goes into effect when nothing could be bound to a parameter ByValue. So we're sitting here looking at a -Name parameter that wants to take pipeline input ByPropertyName. We've given it Process objects. Do those Process objects have a Name property? According to **Get-Member,** they *do indeed!* So the Name property of the input objects will be passed to the -Name property of the cmdlet. The result is that the **Stop-Service** cmdlet will try and stop services *based on their process name.* In many cases, it will be able to do so because a service's name is often the same as the name of its process when the service is running.

So you have to be a bit careful with this business of piping objects from one cmdlet to another—sometimes it'll work better than you think, which might be worse than you want.

Week 1, Day 4: Core Cmdlets

Now that you know how to pipe stuff from one cmdlet to another, you might want to learn some of the cmdlets that can let you manipulate objects in the pipeline. I'm going to briefly introduce these and give you a quick example, but I'm going to expect you to *read the help* to learn more about them—and I'll ask some finishing questions to help encourage that independent research:

- **Sort-Object**, or its alias **Sort**, rearranges objects in memory. Just specify the property you want to sort them by, such as **Get-Process | Sort ID**.
- **Select-Object** does a lot of stuff, and you'll often see its alias, **Select**. For now, focus on its ability to grab just the first or last objects in the pipeline: **Get-Process | Sort VM | Select -first 10**.
- **Measure-Object** counts objects. If you specify a property, you can also have it average the values for that property, assuming it contains numeric values. Its alias is just **Measure**: **Get-Process | Measure vm -average**
- **Import-CSV** and **Export-CSV** read and write Comma-Separated Values (CSV) files, like this: **Get-Service | Export-CSV servicelist.csv**
- **ConvertTo-HTML** creates an HTML table. You'll probably want to write the HTML to a file: **Get-EventLog Security -newest 10 | ConvertTo-HTML | Out-File security-events.htm**

Now's the time for that independent research. How would you:

- Change the sort order of **Sort** to be descending instead of ascending (which is the default)?
- Get the *last* 20 objects from the pipeline using **Select**?
- Change the delimiter of a CSV file to a pipe | character instead of a comma?
- Display not only the average value for processing physical memory but also the minimum and maximum values and the total physical memory used?

The more comfortable you become reading the help, the more capabilities you'll find!

Week 1, Day 5: Configuration Baselines

This is the last tip for your first week of PowerShell, and it's a doozy. First up is a pair of cmdlets that read and write XML-formatted files: **Import-CliXML** and **Export-CliXML**. For example, let's export all the running processes to a file:

```
Get-Process | Export-CliXML baseline.xml
```

Now, launch a couple more processes:

```
Notepad
Calc
Mspaint
```

Now for a fun new cmdlet called **Compare-Object**, or **Diff** as us slow typists like to call him. This cmdlet isn't that good at comparing text files (remember, PowerShell kinda hates text), but it's *awesome* at comparing sets of objects. Consider this command:

```
Diff (Ps) (Import-CliXML baseline.xml)
```

Couple of fun things happening there. First, **PS** is just an alias for **Get-Process**. The really fun thing is the placement of the parameters. You see, I should really have written the command like this:

```
Diff -referenceObject (Ps) -differenceObject (Import-CliXML baseline.xml)
```

This time, I'm including the actual parameter names. But I looked in the help (as I'm sure you did), and saw that -referenceObject is *positional,* and occupies the first position. The -differenceObject parameter is also positional, and occupies position 2. With these *positional* parameters, I don't need to type the parameter name so long as I put the parameter *values* into the correct positions. Thus:

```
Diff (Ps) (Import-CliXML baseline.xml)
```

The parentheses are doing something special. Just like in algebra, they tell the shell to execute whatever is inside the parentheses first. The result of whatever's *inside* the parentheses is passed to the parameter. So the result of **Get-Process** (which is a bunch of process objects) is passed to -referenceObject, and the result of **Import-CliXML baseline.xml** is passed to the -differenceObject parameter. Basically, I'm comparing two sets of processes: the current set and the set that I had exported to a CliXML file earlier.

The results are horrible. Oops. That's actually because *everything about processes is constantly changing*, including their memory use, CPU use, and so on. I'd do better to just compare a single, unchanging property—like name:

Diff (Ps) (Import-CliXML baseline.xml) -property Name

Ah, there are some results. I can now see which objects were present in the left side (the current processes) but not in the right (the baseline). So this is a difference report of my current configuration versus my baseline configuration. Imagine what other types of objects you could export and compare in this fashion!

PowerShell Crash Course Week 2

Hopefully, you're ready for the second week of your crash course. The previous five lessons should have you comfortably running medium-complex commands from the command-line. This week, we're going to focus on fine-tuning those skills and giving you some additional options and capabilities.

Remember, I encourage you to continue exploring beyond this crash course. For example, visit http://windowsitpro.com/go/DonJonesPowerShell to find tips and tricks, 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). Finally, 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.

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 2, Day 1: Variables

So far, we've just been running commands and allowing the output to show up on the screen. But sometimes you might want to preserve command output for a while and re-use it, or even persist some other bit of information. That's where *variables* come in. No, we're not going to be doing programming—variables are just a named storage area in memory. Think of PowerShell's memory space as a warehouse. The warehouse is full of boxes, and we can put stuff into the boxes. To make them easier to find, the boxes each have a name. Those boxes are variables.

Variable names can include almost any character, including spaces, although you'll probably want to stick with letters, numbers, and the underscore character. When you want PowerShell to use the contents of a variable, you precede the variable's name with a $. That's a special character that simply tells the shell, "Hey, what comes next is a variable name—I want you to use the contents of that variable." For example **Computer** might be a variable name, and **$Computer** tells PowerShell that you want to reference its contents. **${This is a variable}** is valid, too. Normally, a white space indicates the end of the variable name, but in this case the variable name "This is a variable" contains spaces. Surrounding the name in curly braces helps keep it all together. I'm not a fan of variable names that contain spaces simply because that syntax is harder to read, but you can do whatever you like.

Read This Bit if You're Experienced with VBScript

At this point, VBScript folks usually ask if there's a way to declare variables in advance. Yes, there is. There's actually a whole raft of -Variable cmdlets, including New-Variable, for that purpose. Note that those cmdlets accept a -name parameter, which specifies the name of the variable. *The variable name does not include the $.* If you ran **New-Variable -name $mine -value 7**, it would attempt to create a new variable *using the contents of $mine as the variable name.*

The next question is *always*: "Is there a way to require advance variable declaration?" As in, "How do I do Option Explicit in PowerShell?" No, there isn't. You can't. Sorry. Yes, I'm aware that this lack can make debugging harder, but the PowerShell team hasn't implemented this. There is something similar—read the help for Set-StrictMode—but it's not the same as Option Explicit.

You put stuff into variables using the = assignment operator:

```
$var = Get-Services
```

You don't need to create the variable in advance; the first time you use it, PowerShell will create it. The previous command retrieves all the service objects and sticks them into $var. Technically, $var is now an array, or collection (in PowerShell, they're close to being the same thing). If you want to display them, just tell the shell to display the variable:

```
$var
```

You can also access individual elements. Try these:

```
$var[0]
$var[1]
$var[-1]
$var[-2]
```

The square brackets contain the zero-based index number of the element you want out of the collection; negative numbers start at the end of the collection. Go on, try it. You'll see.

Variables can also store simple values like strings and numbers—although these are, technically, also objects.

```
$var = 5
$var = 'Hello'
$var2 = "$var World"
$var3 = '$var World'
$var | Get-Member
```

Now try to figure out what $var2 and $var3 contain. Better yet, run those commands in the shell. You'll learn a few things:

- You can easily change the contents of a variable anytime you like. Just because it started out containing services doesn't mean it can't switch to contain a simple string. It's just a box, after all—you can replace its contents.
- In double quotes, PowerShell looks for variables—keyed by the $—and replaces them with their contents. That doesn't happen in single quotes.
- When you pipe a variable to Get-Member, you see information about whatever's *inside* the variable. We don't care about the box itself—we only care about what's *in* the box. Ask any kid at Christmastime—they'll explain that concept to you!

PowerShell stores all of its variables in the Variable drive.

```
Dir Variable:
```

You'll see a lot of the built-in variables that control PowerShell's operation and behavior, too. You can delete variables by using the **Del** command right in that drive, or the **Remove-Variable** cmdlet. Variables don't persist between shell sessions, and the built-in variables (which can be deleted, if you want) always return with their default values when you open a new shell window.

PowerShell keeps track of what kind of object or objects a variable contains, but it doesn't normally enforce any rules around them. In other words, this is valid:

```
$x = 5
$x = "Realtime"
```

You can start out with one type of thing (an integer, in this case) in the box, then replace it with something else (a string). You can, however, *tell* PowerShell what type of data you want a variable to contain. Once you do so, PowerShell will enforce it. Try this:

```
[int]$x = 5
$x = 'Hello'
```

You get an error on the second line because $x can now only contain an [int] and there's no way to convert "Hello" into an integer. Other variable types include [string], [xml], [Boolean], [double], and so on.

Finally, variable names—like cmdlet names—aren't case-sensitive. $X and $x are the same thing.

Week 2, Day 2: Operators and Filtering

PowerShell has the ability to compare bits of information to each other. It does so using a set of comparison operators:

- -eq = equality
- -ne = inequality
- -ge = greater than or equal to
- -le = less than or equal to
- -lt = less than
- -gt = greater than
- -like = wildcard (use * wildcard character)
- -notlike = opposite of -like

Here's how you might use these:

```
$x = 5
$y = 10
$x -gt $y
$y -eq $x

$a = "Hello"
$b = "Hello World"
$a -eq $b
$b -like '*Wo*'
```

Note that string comparisons are case-insensitive; use -ceq, -clike, and -cnotlike for case-sensitive comparisons.

One way to use these comparison operators is in filtering items out of the pipeline. Let's quickly revisit how this all works. I'm going to use some different terminology than in the previous chapter, and this terminology glosses over some technical fine details, but I think it makes the basic concept a lot clearer.

When you run a command like Get-Service, it produces a table of information. Thanks to some formatting defaults provided by Microsoft, however, you only *see* a portion of this table on the screen. To see all of its possible columns, pipe the table to Get-Member:

```
Get-Service | Get-Member
```

Anything in that output that says "Property," including things like "NoteProperty" and "AliasProperty," are table columns. You can use those column names, along with comparison operators, to filter items out of the pipeline. Use the **Where-Object** cmdlet (or its alias, **Where**) to do so:

```
Get-Service | Where { $_.Status -eq 'Running' }
```

Those {curly braces} represent something called a *script block;* in this instance, it's actually something more specific: a *filter block.* You can put basically whatever code you like in there, so long as the final result is either True or False. Within that block *and only within that block,* PowerShell is designed to look for a special $_ placeholder. This is like a "fill in the blank" area on your tax form, and the shell "fills in the blank" with whatever was piped into Where-Object. In this case, then, $_ represents that table of service information.

The period tells the shell that you don't want to deal with the entire table but instead want to access the information in a single column. You then follow the period with the column name: Status. So I'm comparing the Status column to the string "Running;" table rows that match will be retained, and ones that don't will be dropped. Try running the command to see the results.

You can do complex comparisons, too:

```
Get-WmiObject -class Win32_Service | Where { $_.State -ne 'Running' -and
$_.StartMode -eq 'Auto' }
```

This is grabbing (from WMI, this time) all the services that are set to auto-start but which aren't yet started. The -and operator combines the two comparisons. There are also an -or operator and a -not operator that reverse the logic. Keep in mind that $_ isn't globally present throughout the shell: It's only valid in places where PowerShell is explicitly looking for it, and this is one of the few places where the shell does so.

This syntax can be a bit difficult at first, but *this is one of the most important concepts in the shell.* Don't proceed until you're comfortable with it. To practice, try some of these tasks:

- Get a list of processes that have an ID greater than 1000
- Get a list of event log entries (Get-EventLog is the cmdlet to start with) that have an event ID greater than 1500
- Get a list of processes that have "svchost" in their name
- Get a list of services that have "win*" in their name

Once you can accomplish those tasks, you're good to go. If you get stuck, drop by http://connect.concentratedtech.com where you can post a question directly to me. When you register, you'll be asked where you took a class with me—enter "Realtime Publishers" for that field.

Week 2, Day 3: Formatting

So far, we've been happy to use the formatting that the shell gives us by default, although in the first chapter of this book, I did show you a few alternatives for formatting output as tables and lists. Now, we need to jump in and take full control. This can get a bit complex, so bear with me. First, know that you have three basic formatting cmdlets that you'll use most of the time:

- **Format-Table**, which produces a columnar table
- **Format-List,** which produces a simpler list
- **Format-Wide,** which displays a single piece of information in a multi-column list

You *should read the help on all of these* for a quick overview of what each can do. Format-Wide is the simplest: You can specify the property (table column) you want displayed, and it defaults to the "name" property because most objects have a Name property. You can also change the number of columns displayed or specify -autosize to let it display as many as it thinks it can. *Read the help* and see if you can figure out how to display a four-column list of process names. Go on, I'll wait for you.

Did you get it? Well, I'm not giving away any answers. Again, if you're stuck, hop on http://connect.concentratedtech.com and I'll help you out.

Format-List is probably the next-most-complicated. With it, you can specify a *list* of properties to display, and that list can include wildcards. For example, try these:

```
Get-Process | Format-List *
Get-Process | Format-List *Memory*
Get-Process | Format-List Name,ID,VM
```

Get the idea?

Format-Table is the most fun. Some of the options I'm going to explore also exist for Format-List, but I think they're more impressive when used with Format-Table.

First, be aware that Format-Table normally tries to fill your entire screen with text, edge to edge, even if that means putting a lot of space in the middle. You can, however, tell it to try and automatically size each table column for whatever that column needs to contain. Notice the difference in these:

```
Get-Service | Format-Table Name,Status
Get-Service | Format-Table Name,Status -autosize
```

If you've sorted your objects, you can have Format-Table group them. Each group gets a new set of column headers. For example:

```
Get-Service | Sort Status | Format-Table Name,Status -autosize
-groupby Status
```

Neat, right?

You can also create *custom columns*. These might simply translate one kind of value to another. For example, the Win32_LogicalDisk class includes a Size property that contains a disk's size in bytes; we could make that megabytes in our output by adding a custom column. The syntax for doing so is a bit complicated, though. You need to create something called a *hashtable.* These are also known as dictionary objects. They can contain multiple entries, and each entry consists of a key and a value (or, as I like to think of them, a *word* and a *definition*). Making one looks like this:

```
@{'Key'='Value';'Word'='Definition'}
```

That hashtable has two items. The items are separated by a semicolon, the whole thing is contained in {curly braces}, and an @ sign tells the shell that we're defining a hashtable. When making custom columns in Format-Table, you're not quite so freeform. You have to specify exactly two keys: Label (which can also be called Name), and Expression. These can be shortened to just "l" and "e," which are a lot easier to type. The Label is what you want your custom column to be titled, and the expression is a script block that calculates the column's content. Within that script block—just as with Where-Object, which I showed you earlier in this chapter—you can use the $_ placeholder to refer to the remainder of the current table row.

Here's an example. This is all meant to be typed on one line, by the way.

```
Get-WmiObject -class Win32_LogicalDisk -filter 'DriveType=3' |
Format-Table DeviceID,@{l='Size(MB)';e={$_.Size / 1MB -as [int]}}
```

Yeah, I know—complicated. Told you so. Focus on that hashtable: I've set a label of "Size(MB)," which is straightforward enough. The expression ("e") goes into a {script block} rather than in quotation marks. Within that script block—*and only within that script block*—you can use that $_ placeholder to refer to the row of information. Follow it with a period, then the piece of information you want—"Size." I divided by 1MB (yup, PowerShell recognizes KB, MB, GB, TB, and PB). I used the -as operator to convert the result to an [int] (integer), forcing it to drop the fractional portion of the result so that I get a whole number. I'll leave you with that. Play with it, practice with it. Hit me up on http://connect.concentratedtech.com if you get stuck.

One caution: The output of a Format- cmdlet is very specialized, and it can only be used by some very specific PowerShell cmdlets, which I'll discuss next. In short, the Format cmdlet generally needs to be *the last thing on the command line*. You can't do this:

```
Get-Service | Format-Table | ConvertTo-HTML > services.html
```

Although I invite you to try. The results may surprise you.

Week 2, Day 4: Getting "Out"

Once you've formatted data, you might be interested in looking at it. By default, anything left in the pipeline at its end will be sent to a built-in, hardcoded cmdlet called **Out-Default**. In the PowerShell console host and ISE, Out-Default simply forwards to **Out-Host**. Out-Host is the cmdlet responsible for putting information on the screen ("host" means "screen," here).

There are other Out cmdlets you can play with: **Out-File** and **Out-Printer** are two common ones. Together, these four Out cmdlets have one thing in common: *They only consume the output of a Format cmdlet.* That is, if you run this:

```
Get-Service
```

What's really happening under the hood looks something like this:

```
Get-Service | Out-Default | Format-Table | Out-Host
```

That's not exactly what's happening, but it's an illustration. Anyway, I told you earlier that a Format cmdlet, if used, needs to be the last thing on the command line. The only exception is that a Format cmdlet can be followed by Out-Host, Out-Printer, or Out-File. In fact, these two commands are identical because > is just an alias (sort of) to Out-File:

```
Dir > files.txt
Dir | Out-File files.txt
```

Read the help for Out-File, by the way. You'll find a number of really useful options.

There's also **Out-GridView**, which only works if you have the PowerShell ISE and .NET Framework 3.5 installed. Try this:

```
Get-Process | Out-GridView
```

I'll leave you with that. Experiment with some of the Out- cmdlets and see what you can come up with.

Week 2, Day 5: Extending the Shell

Stop thinking about PowerShell for a second and start thinking about the MMC. When you open a new, blank MMC, it isn't very useful, is it? You have to add snap-ins to it in order to add actual functionality. You can add as many snap-ins as you like, from whatever administrative tools you've installed on your computer.

PowerShell is the same way. Now, there's a bit of confusion going around about this. Install the Exchange Server admin tools and you'll get an "Exchange Management Shell" icon. SharePoint does something similar. Windows Server 2008 R2 has an "Active Directory" shell icon. *These aren't different versions of PowerShell.* Like the MMC, these are just icons that load the same old PowerShell, with a particular extension pre-loaded. You can also load those same extensions into the normal shell yourself, and if you open one of these "pre-loaded" shells, you can still load whatever other extensions you like. It's your choice.

When you do load an extension, it only stays loaded for the duration of that shell session. When you open a new shell window, you're starting from scratch. A bit later, I'll introduce you to profiles, which give you a way to have the same extensions loaded in every new shell window. For now, you need to know that there are two ways of extending the shell:

- PSSnapins (or "snap-ins") is the old-school, v1 way of doing it. Snap-ins have to be installed (generally, install the administrative tools for a product and you'll get the snap-in). Run **Get-PSSnapin -registered** to see which ones are installed; with that list of names, you can run **Add-PSSnapin** to load a particular snap-in by name. You'll still find new snap-ins being produced—even Windows Server 2008 R2 shipped with a new snap-in or two.
- Modules are the new, v2 way of extending the shell. Run **Get-Module -list** to see a list of what's available on your computer, and **Import-Module** (along with the module name) to load a module into memory.

Once you've loaded an extension, you can ask the shell to show you what was added:

- Run **Get-Command -pssnapin *name*** to see the commands added by the PSSnapin you specified; replace *name* with the snap-in's actual name.
- Run **Get-Command -module *name*** to see the commands added by the module you specified.

The Windows 7 Remote Server Administration Toolkit (RSAT) adds a bunch of modules, and Windows 7 ships with a few pre-installed. Windows Server 2008 R2 installs modules along with their associated roles, so a domain controller will have the ActiveDirectory module. Third parties can also produce modules and snap-ins:

- Quest.com/PowerShell—alternate AD cmdlets
- CodePlex.com/powershellcx—open-source community extensions

I maintain a directory of the higher-quality stuff at http://ShellHub.com. Drop by and see what tickles your fancy.

Weekend Wrap-Up

If you're sticking with the one-lesson-per-day regimen, then you're probably ready for the weekend. Take a couple of days off! The next installment will have another week of lessons for you. Until then, keep practicing what you've learned!

By the way, if you got this chapter as a pass-along from a friend, thanks for reading it. Hop on http://nexus.realtimepublishers.com for more chapters from this book, and for a lot of other free books. Registration is required, but you'll be able to download PDF versions of all their books at no charge. Speaking as an author, that's very important: Your registration lets the publisher know that you're interested in this book, and that keeps the publisher engaging authors like myself to write more for you. I appreciate it.

PowerShell Crash Course Week 3

Hopefully, you're ready for week 3 of your crash course. The previous ten lessons have covered the majority of Windows PowerShell's core techniques and patterns. This week we're going to start learning about some of its cooler embedded technologies, building on those techniques and patterns.

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 3, Day 1: Remote Control

One of the coolest new features in Windows PowerShell v2 is its built-in remoting. With it, *every* command can be run on one or more remote computers—even simultaneously—without the commands' authors having to do any extra work to make it possible.

Pre-Requisites and Setup

First, you need to get remoting enabled. It requires that PowerShell v2 be installed on each and every computer that you want to remote to. Then run **Enable-PSRemoting** on those computers to turn on the ability to receive incoming connections. You can also control that through GPO: Run **help about_remote_troubleshooting** for details on how to set that up and for help on topics like connecting non-domain computers, remoting across a proxy server or firewall, and so on.

Authentication, by default, is done using Kerberos—so your password stays safe. By default, you'll remote using whatever credential was used to launch PowerShell—normally, it'll need to be an Administrator account. The service involved in remoting is Windows Remote Management (WinRM), and the protocol is Web Services for Management (WS-MAN), which rides atop HTTP. Enable-PSRemoting will set up the necessary exceptions in the Windows Firewall for you.

1:1 Remoting

If you just need to do something with one computer, then you can remote to it interactively—kind of like SSH in Unix. PowerShell does provide encryption, and you can have the entire connection run over HTTPS (rather than the default HTTP) for greater security. Just run **Enter-PSSession -computer *computername*** to start a session; you'll see the PowerShell prompt change appropriately, and you'll be running commands on the remote computer. When you're done, run **Exit-PSSession** to return to your local prompt and close the remote connection.

Enter-PSSession offers a number of options. Review the help and see if you can figure out how you would:

- Add SSL (HTTPS) to the connection
- Specify an alternate IP port (used if you've changed the port that WinRM listens to on the remote machine)
- Specify an alternate credential for the connection

1:Many Remoting

Managing one computer at a time can be useful, but it's not *automation.* What's often better is the ability to have the same command run on *multiple* computers, all at once. You do that with **Invoke-Command.** Here's a simple example using three computers:

```
Invoke-Command -scriptblock { Get-EventLog -Log Security -newest 100 }
  -computername server1,server2,server3
```

There's a lot of ways to get computer names into the -computername parameter. What I did is provide a comma-separated list, which PowerShell sees as an array of items. Another way is to read names from a text file, which contains one name per line:

```
... -computername (get-content names.txt)
```

The parentheses force the shell to execute the Get-Content command first, then feeds its output to the parameter. Parentheses are a useful trick, and there are several ways in which you can use them. Perhaps you have a CSV file that contains a column named Server? Here's how you'd use that column:

```
... -computername (import-csv servers.csv | select -expand Server)
```

The -expand parameter of Select-Object tells the command to grab the *contents* of the property designated—in this case, the Server column of your CSV, and feed just those values as the command output. That gets stuck onto the -computername parameter, and voila! You can use the same trick to query computers from Active Directory (AD), assuming you've loaded the ActiveDirectory module:

```
... -computername (get-adcomputer -filter *
  -searchbase "ou=servers,dc=mycompany,dc=local" | select -expand Name)
```

The -scriptblock parameter of Invoke-Command is where you put the commands you want to run remotely; multiple commands can be separated by a ";" semicolon. You can feed in an entire pipeline, and you should have as much processing done remotely as possible. For example, this:

```
Invoke-Command -scriptblock { ps | sort vm -desc | select -first 10 }
  -computername server1,server2,server3
```

Is better than this:

```
Invoke-Command -scriptblock { ps } -computername server1,server2,server3 |
  Sort vm -desc | select -first 10
```

The second example has far too much data being transmitted from the remote machines—and it won't produce the same results, anyway. See if you can figure out why (if not, drop by http://connect.concentratedtech.com and ask—I'll be happy to give you a clue. If you're registering for the first time, enter "Realtime" as the place where you took a class from me and I'll make sure your registration is approved).

By default, Invoke-Command talks to as many as 32 computers at once. Give it more than that, and the extras get queued up and completed in order. You can change that with the -Throttle parameter. Other parameters you should look for (read the help—no hints from me!):

- How to specify an alternate credential
- How to specify an alternate IP port
- How to enable SSL (HTTPS) for the connections
- How to specify a script file path instead of an individual command

Notice, too, that the results of Invoke-Command have an extra property attached. The PSComputerName property contains the name that each result came from so that you can keep track of them. You could sort and group that output by computer name once the output is back to your computer:

```
Invoke-Command -scriptblock { Dir c:\ } -computername (gc names.txt) |
  Sort pscomputername | format-table -groupby pscomputername
```

Also, keep in mind that you can only invoke commands *that are available on the remote computer.* Let's say all of the remote computers are Windows Server 2008 R2 machines, and you want to see what Best Practices Analyzer (BPA) models are available on each. To do that, each remote computer needs to load the BestPractices module:

```
Invoke-Command -scriptblock { Import-Module BestPractices; Get-BPAModel }
  -computername server1,server2,server3
```

That would load the module on each, then execute a command from that module. When the command finishes executing, the connection will close and the module will unload automatically.

Re-Using Connections

When you give a computer name (or computer names) to Invoke-Command or Enter-PSSession, the commands create a connection, then automatically tear it down when you're done with it. That means every time you open a new connection, you might have to specify credentials, alternate ports, and other options—which can be tedious.

Another option is to create a reusable connection, called a PSSession. When you do so, you'll probably want to save the session (or sessions) in a variable because that makes it easier to re-use it. For example:

```
$webservers = New-PSSession -computername www1,www2,www3
 -credential Administrator
```

When you're ready to use the session, just provide it to the -session parameter of Invoke-Command:

```
Invoke-Command -script { Get-Process } -session $webservers
```

Enter-PSSession only works with a single computer, so you can use an index number to specify which of those sessions you want. For example, the second session (www2) would be:

```
Enter-PSSession -session $webservers[1]
```

To close the sessions:

```
$webservers | Remove-PSSession
```

That also happens automatically when you close your shell. Leaving a session open does occupy some processor and memory overhead on both your machine and the remote one, but it's not a massive amount. Just don't get every admin in the habit of holding a bunch of sessions open, and you'll be fine.

Week 3, Day 2: In the Background

You may often have times when you want to kick off some long-running task, but you don't want to wait for it to complete. PowerShell provides background jobs as a way of moving a task into the background, where the shell will continue to execute and monitor it. The shell will also cache any results that the job generates, enabling you to receive those results once you're ready.

Starting a Job

There are three ways to start a job:

- If it's a local task that won't connect to remote computers, use the **Start-Job** cmdlet. Give it a -command parameter, which tells it what commands to run in the background:

```
Start-Job –command { Get-EventLog –log Security }
```

- Get-WmiObject has an –AsJob parameter, which forces the WMI processing into the background—especially useful when dealing with many remote computers.

```
Get-WmiObject -class Win32_BIOS -computer server1,server2 –asjob
```

- Invoke-Command also has an -AsJob parameter, which forces it to invoke commands in the background—again, useful when you're running against multiple remote computers:

```
Invoke-Command -scriptblock { Get-EventLog -log Security }
 -computername server1,server2,server3 -asjob
```

By default, you'll create a job with a generic name like "Job1." Both Start-Job and Invoke-Command provide parameters (-name and -jobname, respectively) that let you specify a custom job name—often making it easier to manage multiple jobs.

Managing Jobs

Run **Get-Job** to see a list of jobs and their states—Running, Completed, Failed, and so forth. Note that every job has an ID number, along with a name. You can get the status for a particular job by using that ID or name. I like to pipe the results to Format-List so that I can see all of the job's details:

```
Get-Job -name MyJob | Format-List -property *
```

A key thing to know is that the shell always creates a "parent" job, plus one "child" job for each computer. So a job created by using Start-Job will have a parent and one child—which is the local computer. An Invoke-Command job targeting four computers will have the parent job and four children. The Format-List trick will allow you to see the names of each child job—enabling you to get those jobs individually:

```
Get-Job -name Job2 | Format-List -property *
```

When you're done with a job, you can remove it to clean up the list:

```
Get-Job -name MyOldJob | Remove-Job
```

There's also Stop-Job, to kill a stuck job, and Wait-Job, which is useful when a script needs to start a job and then wait until it completes before running more commands.

Getting Results from Jobs

When a job is finished, it'll likely have some results for you. Here's how to get them:

```
Receive-Job -id 4
```

Or

```
Receive-Job -name MyDoneJob
```

By default, the received results are delivered to you and then no longer cached—meaning you can't receive them again. If you want to leave them cached in memory, add the -Keep parameter to Receive-Job.

Week 3, Day 3: Implicit Remoting

Microsoft has always had versioning challenges. For example, Windows Server 2008R2 ships with a new ActiveDirectory module for managing AD—but that module can only run on 2008R2 and Windows 7. What if you're stuck on XP? *Implicit remoting* is designed to help solve the problem.

Here's the scenario: You have a 2008R2 domain controller, which has the ActiveDirectory module already installed (it comes along with the domain controller role). You enable remoting (Enable-PSRemoting) on that domain controller, then settle down in front of your Windows XP machine that has PowerShell v2 installed. Run these commands, assuming DC2008 is the name of that domain controller:

```
$session = New-PSSession -computername DC2008
Invoke-Command -script { Import-Module ActiveDirectory } -session $session
Import-PSSession -session $session -module ActiveDirectory -prefix Rem
```

Now, all of the commands in the ActiveDirectory module are available as local commands on your Windows XP machine. A prefix, "Rem," has been added to each command's noun, helping remind you that the commands will actually *execute* remotely. You can even ask for help on them. For example, all of these will now run fine:

```
Get-RemADUser -filter *
Help New-RemADUser
```

Neat trick, right? The idea is that the server providing the functionality can now also provide the administrative capability, and you don't need to install every single admin tool locally on your computer. This also helps mitigate version mismatches by not caring what OS you're running on your workstation—tools simply run on the server.

Week 3, Day 4: Making a Simple Reusable Command

Let's say you've created a really kick-butt command that accomplished some great stuff. It's a lot of typing, but you love the way it works:

```
Get-WmiObject -class Win32_LogicalDisk -filter "DriveType=3"
 -computername server1,server2,server3 | Select-Object DeviceID,
 @{n='ComputerName';e={$_.__SERVER}},
 @{n='Size(GB)';e={$_.Size / 1GB -as [int]}},
 @{n='FreeSpace(MB)';e={$_.FreeSapce / 1MB -as [int] }}
```

You want to share that with some other techs in your organization, but you don't want them to have to retype all that—or even edit it because you *know* they'll mess up something. No problem. Put this into a text file with a .ps1 filename extension:

```
Function Get-DriveInventory {
 Param($computername)
 Get-WmiObject -class Win32_LogicalDisk -filter "DriveType=3"
  -computername $computername | Select-Object DeviceID,
  @{n='ComputerName';e={$_.__SERVER}},
  @{n='Size(GB)';e={$_.Size / 1GB -as [int]}},
  @{n='FreeSpace(MB)';e={$_.FreeSapce / 1MB -as [int] }}
}
```

The three lines in boldface are what I added, along with the boldfaced $computername in the command itself. Let's say you put this in a file named c:\scripts\utilities.ps1. Your other admin friends could then add the following to their PowerShell profile (run **help about_profiles** for more info):

```
. c:\scripts\utilities
```

That leading dot will load the contents of Utilities.ps1 into the shell's global scope, making your command available just like any other. To use it:

```
Get-DriveInventory -comp server-r2
```

The shell even supports abbreviating the -computername parameter, as shown here. Neat, right?

Week 3, Day 5: SELECTing

The example from the previous section utilized a little-known feature of the Select-Object cmdlet. This command has four really useful features:

- You can tell it what properties you want to keep for an object, helping to whittle down output to exactly what you want to see:

```
Get-Process | Select -property ID,Responding,VM,PM
```

- You can add custom properties, using a hashtable—just like you did with Format-Table in Chapter 2. The hashtable needs two keys: "N" is the name of the new property, and "E" is the expression used to create the new property's values:

```
Get-WmiObject -class Win32_LogicalDisk -filter "DriveType=3"
 -computername server1,server2,server3 | Select-Object -property DeviceID,
 @{n='ComputerName';e={$_.__SERVER}},
 @{n='Size(GB)';e={$_.Size / 1GB -as [int]}},
 @{n='FreeSpace(MB)';e={$_.FreeSapce / 1MB -as [int] }}
```

- You can also select a subset of the objects piped in—grabbing the first or last however many:

```
Get-Service | Select -first 10
```

- You can tell it to just grab the *values* from a given property. This forces the command to output simple values instead of a more complex object with one or more properties:

```
Get-ADComputer -filter * | Select -expand Name
```

Give these techniques a try with various other commands and see what happens.

Week 3 Wrap-Up

This week, you've learned about powerful embedded technologies within PowerShell—and you've also learned some new patterns. For example, the various ways in which I showed you to feed computer names to the -computername parameter of Invoke-Command are all patterns you'll find uses for with many other cmdlets. Pay attention to these different patterns, and try to think of ways in which you might use them elsewhere within the shell.

I'll leave you with a final super-complex example. Look through this carefully, and see if you can figure out why it does what it does. This is safe to run on a production system or your workstation, too, so that you can see the results. How would you modify this to run against a remote computer?

```
Get-WmiObject -class Win32_OperatingSystem | Select-Object
 @{n='OSVersion';e={$_.Caption}},
 @{n='SPVersion';e={$_.ServicePackMajorVersion}},
 @{n='OSBuild';e={$_.BuildNumber}},
 @{n='ComputerName';e={$_.__SERVER}},
 @{n='BIOSSerial';e={
  (Get-WmiObject -class Win32_BIOS -computername $_.__SERVER |
  Select -Expand SerialNumber)
 }} | Format-Table -property * -wrap
```

Good luck!

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!