The Windows PowerShell Owner's Manual: Version 2.0

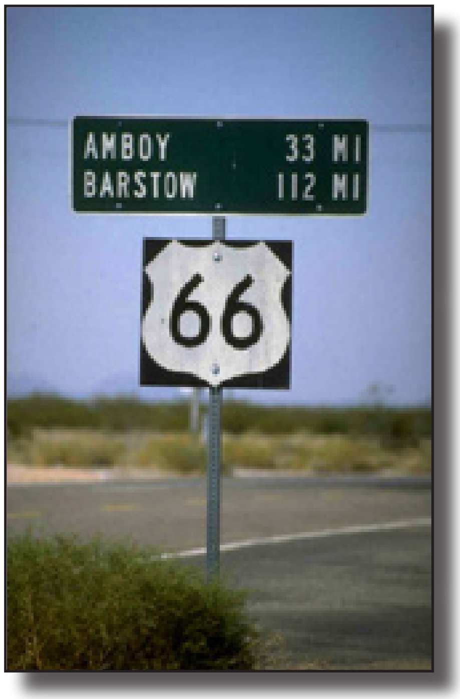
Jean Ross and Greg Stemp
Microsoft Communications Server UA

Contents

About This Owner's Manual

A couple of years ago we were asked to lead an instructor-led lab at TechEd in Orlando, Florida.

Okay, that's not entirely true. We wanted to go to TechEd to meet with customers and hand out a bunch of fun stuff, but our managers back then said we couldn't go unless we got in for free as speakers. Since our area of expertise was in system administration scripting and Windows PowerShell 1.0 was still very new, we sent a proposal to the TechEd organizers to do a pre-conference seminar. Why a pre-conference seminar? Well, because it takes more than a one-hour talk to teach people Windows PowerShell, and if we were going to teach PowerShell we were going to do it right. Well, as it turned out, the pre-conference seminars were all booked up for that year. But the TechEd organizers had another proposal for us: a 75-minute instructor-led hands-on lab.



Did we mention that you can't teach Windows PowerShell in an hour? But an hour and 15 minutes? Hey, no problem!

Whatever it takes to get in the door, right?

Believe it or not, the labs wound up being pretty successful, and even a lot of fun. But because we couldn't do a full seminar, and because we wanted people to have something to remind them of everything they learned (and then some) after they'd gone home, we came up with a little thing we called the **Windows PowerShell Owner's Manual**. This is that manual.

Well, okay, once again that's not entirely true. This is the *new-and-improved* Owner's Manual. For starters, it's been updated for Windows PowerShell 2.0. We've also added bits of new content here and there. We hope you enjoy it and maybe even learn a little something from it. If you have any questions, feel free to ask; we'll do our best to try and answer them for you.

Jean Ross

Jean Ross (jeanros@microsoft.com)
Microsoft Communications Server UA

Greg Stemp

Greg Stemp (gstemp@microsoft.com)
Microsoft Communications Server UA

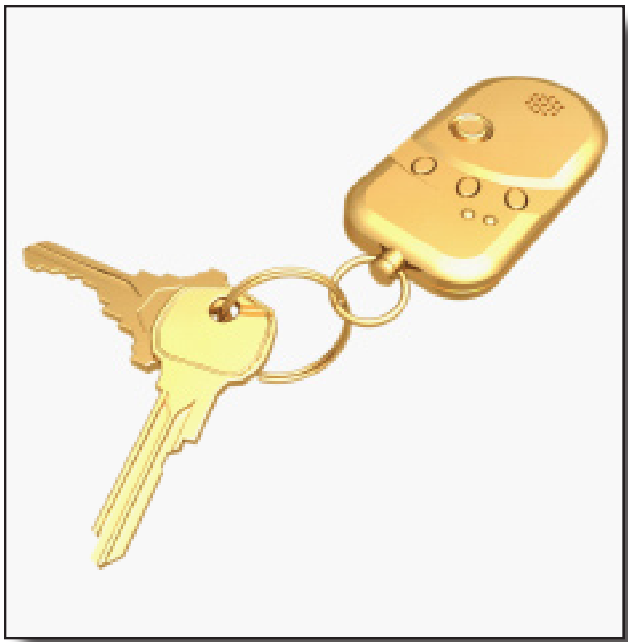
Windows PowerShell Terminology

Cmdlets (pronounced "command lets") are the core commands used in Windows PowerShell; they are analagous both to command-line tools and to WMI classes or COM objects. (How can they be analgous to both command-line tools and WMI classes? Well, like command-line tools they can be called and run from the command prompt; like WMI classes they can be accessed from scripts.) Cmdlet names are typically composed of two parts: a verb (like **Get**) and a noun (like **ChildItem**), with a hyphen between the two. Cmdlet names also tend to be singular: you use Get-EventLog rather than Get-EventLog***s***. To see a list of cmdlets available to you simply type **Get-Command -CommandType cmdlet** from the Windows PowerShell prompt.

```
Get-ChildItem   -path   C:\Scripts
```

Parameters represent additional information passed to a cmdlet. In PowerShell you can have both "named parameters" and "positional parameters." A named parameter is simply the parameter name (e.g., -path) followed by the parameter argument. (Note that parameter names always begin with a hyphen.) A positional parameter is simply a parameter that always occurs in a specified spot; as such, you do not have to specify the parameter name. Because -path is a positional parameter, the command shown on this page could also be written like this: **Get-ChildItem C:\Scripts**.

An **argument** is a value passed to a parameter. (You must enclose a string value in double quotes if that value contains a blank space. Otherwise double quotes are optional.) Many parameters accept wildcard characters; to return a collection of all the .TXT files in the folder C:\Scripts we can use this command: **Get-ChildItem C:\Scripts\*.txt**.



Getting Started

Can't figure out how to get started in Windows PowerShell? Hmmm, if only we could find an article titled ***Getting Started*** *....*

First you put the key in the ignition… No, wait, that's really not the best place to start. That's not the first thing you should do when you're learning about your car, and it's certainly not the first thing you should do when you're just getting to know Windows PowerShell. So where *should* you start? How about right here…

About Windows PowerShell

For most things that come with an Owner's Manual you probably knew what that thing was before you got it; if you go out to buy a cell phone you probably have a pretty good idea what a cell phone is and what it does. And like those other things, you probably have some idea what Windows PowerShell is. It's just a command window, right? Oh, no, it's actually a scripting language. Wait, hang on; it's a set of command-line tools….

If you're sufficiently confused as to what, exactly, Windows PowerShell is, we'll clear up some of that confusion right now: It's all of those things we just mentioned. Windows PowerShell is a command window with its own built-in commands (called cmdlets) and its own scripting language. If that sounds powerful, well, that's because it is. If it sounds just a little scary, well …that's what the Owner's Manual is for. Once you get familiar with Windows PowerShell, learn what all the parts and accessories are and how they work, it's really not scary at all. As a matter of fact, it turns out it's one of the most useful – uh, *things* that a Microsoft Windows system administrator can have at his or her disposal.

A Brief History

If you're not a history buff you can skip this section. But because some people find this interesting, we threw it in.

When Windows PowerShell was first conceived, it was supposed to be a replacement for the age-old Windows command window (cmd.exe). For example, there were going to be improvements such as being able to use Ctrl+C and Ctrl+V to copy and paste. Well, that part never actually happened. Instead, over a period of several years it morphed into something else. For a while it was going to be a way for UNIX administrators to feel more comfortable using Windows. UNIX administrators typically spend more time at the command line than Windows administrators, who tend to rely more on the graphical user interface (GUI) provided by Windows. So PowerShell was going to provide a more robust command-line experience for UNIX administrators.

At what point PowerShell turned into a scripting language and a full command-line experience is a little fuzzy. Windows PowerShell had been in development for several years and was struggling to take hold within the Windows group. Then along came the Microsoft Exchange team. This team was looking for a better way to manage Exchange from outside the GUI. They decided to work with the PowerShell team to produce an Exchange-specific implementation of Windows PowerShell. Following the Exchange team's example, several other Microsoft Server products began adopting Windows PowerShell. The product finally shipped with Windows as an add-on to Windows Server 2008, and became integrated into the system as of Windows 7. The rest, as they say, is history.

Windows PowerShell 1.0 was released in 2006. The Windows 7 release, in 2009, is version 2.0 and includes some much-anticipated functionality, not the least of which is remote management.
And that's the end of our history lesson. Glad you stuck around for it?

Oh. Well then, let's move on.

All About Cmdlets

We're going to start our discussion of PowerShell by talking about cmdlets (pronounced command-lets). Cmdlets are really what make PowerShell work; without cmdlets, PowerShell is little more than cmd.exe. (It *is* more, but definitely little more.) Before we explain what cmdlets do and how they work, we need to explain the cmdlet naming convention.

One thing that distinguishes PowerShell cmdlets from standard command-line tools is the way they're named. Every PowerShell cmdlet consists of a verb followed by a dash followed by a noun. This combination not only identifies something as a cmdlet but it will typically give you a pretty good idea of what the cmdlet does. For example, here's a cmdlet you'll probably use quite often:

```
Get-Help
```

Notice the verb (Get), the dash (-), and the noun (Help). And what does this cmdlet do? That's right, it gets you some help. (And no, unfortunately it *doesn't* automatically call up a PowerShell expert and send them straight to your office. It simply displays helpful information to the command window. We'll talk more about Get-Help in just a bit.)

Okay, quick test to see if you were paying attention. Which of these is a PowerShell cmdlet?

```
GetProcess
EventLogFinder
Eat-Dinner
Ipconfig
Get-Service
Directory-Read
```

If you chose Get-Service you're right. Extra credit for anyone who noticed that Eat-Dinner follows the PowerShell naming convention and therefore *could* be a cmdlet.

Note: The PowerShell team does have pretty strict guidelines as to what verbs can and can't be used. As of this writing, Eat was not on the list of approved verbs.

As we've implied (or at least as we meant to imply), a cmdlet carries out a specific action. We already mentioned the Get-Help cmdlet, which displays help text. Here's another example:

```
Get-Service
```

Type that at the command prompt and you'll see output similar to this:

```
Status   Name               DisplayName
------   ----               -----------
Stopped  AdtAgent           Operations Manager Audit Forwarding...
Running  AeLookupSvc        Application Experience
Stopped  ALG                Application Layer Gateway Service
Running  AppHostSvc         Application Host Helper Service
Stopped  Appinfo            Application Information
Stopped  AppMgmt            Application Management
Stopped  aspnet_state       ASP.NET State Service
Running  AudioEndpointBu... Windows Audio Endpoint Builder
Running  AudioSrv           Windows Audio
Running  BFE                Base Filtering Engine
Running  BITS               Background Intelligent Transfer Ser...
Stopped  Browser            Computer Browser
Running  CcmExec            SMS Agent Host
Running  CertPropSvc        Certificate Propagation
Stopped  clr_optimizatio... Microsoft .NET Framework NGEN v2.0....
Stopped  clr_optimizatio... Microsoft .NET Framework NGEN v2.0....
Stopped  COMSysApp          COM+ System Application
Running  CryptSvc           Cryptographic Services
Running  CscService         Offline Files
…
```

As you can see, this cmdlet retrieves a list of all the services on the local computer, along with certain information about those services (such as status).

Using Cmdlet Parameters

It's possible there are a lot of services on your local computer. What if you're interested in only one of those services? You don't need a big long scrolling list of all the services, you just want to see that one. For example, let's say you want to find out if the Windows Update service is running on your computer. You could type Get-Service at the command prompt, hit Enter, then search through the list for the row containing a DisplayName of Windows Update. But there's an easier way:

```
Get-Service -DisplayName “Windows Update”
```

What we've done here is pass a *parameter* to the Get-Service cmdlet. A parameter is a way of handing additional information to a cmdlet. In this case we've used the –DisplayName parameter, followed by the value “Windows Update” to tell the Get-Service cmdlet to get only those services with a DisplayName property equal to Windows Update. (The quotes around Windows Update are required only because the string contains a space.)

Notice one very important thing: the parameter name begins with a dash (-). All cmdlet parameters begin with a dash. In addition, the value you want to assign to the parameter always immediately follows the name of the parameter.
Here's what we get back from our Get-Service cmdlet with the parameter -DisplayName “Windows Update”:

```
Status   Name               DisplayName
------   ----               -----------
Running  wuauserv           Windows Update
```

We can now easily see that our Windows Update service is Running. (Phew!)

Another way we could have accomplished this same thing would have been to pass the –Name parameter and specify the Name value (which is different from the DisplayName) we're looking for:

```
Get-Service -Name wuauserv
```

And once again, here's what we get back:

```
Status   Name               DisplayName
------   ----               -----------
Running  wuauserv           Windows Update
```

Still running.

Note: We've been showing all our cmdlets and parameters in mixed case. And that's fine: PowerShell is *not* case-sensitive. Type this at the command prompt and your results will be the same:

```
get-service –name wuauserv
```

Or even this:

```
geT-sERVICE –naMe WUAuserv
```

Wildcards

Another thing you can do in PowerShell to further refine your results is to use wildcards. The wildcard characters in PowerShell are the asterisk (*), representing one or more characters, and the question mark (?), representing a single character. For example, suppose we want to find all the services whose DisplayName starts with the word Windows. All you need to do is include the wildcard:

```
Get-Service -DisplayName windows*
```

Depending on which services you have on your computer, you'll get back something like this:

```
Status   Name               DisplayName
------   ----               -----------
Running  AudioEndpointBu... Windows Audio Endpoint Builder
Running  AudioSrv           Windows Audio
Running  Eventlog           Windows Event Log
Stopped  FontCache3.0.0.0   Windows Presentation Foundation Fon...
Stopped  idsvc              Windows CardSpace
Running  MpsSvc             Windows Firewall
Stopped  msiserver          Windows Installer
Running  RapiMgr            Windows Mobile-based device connect...
Stopped  SDRSVC             Windows Backup
Running  stisvc             Windows Image Acquisition (WIA)
Stopped  TrustedInstaller   Windows Modules Installer
…
```

Notice that the DisplayName in each row begins with the string "windows". The string "windows*" tells the cmdlet to get all services where the DisplayName begins with the characters *windows* followed by one or more other characters.

You can also put more than one wildcard character within your value:

```
Get-Service -DisplayName *audio*
```

Here we're simply saying "return all services where the DisplayName starts with any character or characters, contains the string *audio*, then ends with any character or characters." Here's what we got back on our test machine:

```
Status   Name               DisplayName
------   ----               -----------
Running  AudioEndpointBu... Windows Audio Endpoint Builder
Running  AudioSrv           Windows Audio
Stopped  QWAVE              Quality Windows Audio Video Experience
```

Multiple Parameters

Now that you know how wildcards work, that will help us demonstrate the next thing you need to know about parameters: multiple parameters. Depending on the cmdlet and the parameters available to it, you can specify more than one parameter at a time. For example, suppose we want to display all the services with a name beginning with windows (remember we saw how to do that: -DisplayName windows*). But wait: we don't really want *all* those service. Instead, we want to exclude all those that contain the word audio in the Name or DisplayName. Here's how we do that:

```
Get-Service -DisplayName windows* -Exclude *audio*
```

And here's what we get back:

```
Status   Name               DisplayName
------   ----               -----------
Running  Eventlog           Windows Event Log
Stopped  FontCache3.0.0.0   Windows Presentation Foundation Fon...
Stopped  idsvc              Windows CardSpace
Running  MpsSvc             Windows Firewall
Stopped  msiserver          Windows Installer
Running  RapiMgr            Windows Mobile-based device connect...
Stopped  SDRSVC             Windows Backup
Running  stisvc             Windows Image Acquisition (WIA)
Stopped  TrustedInstaller   Windows Modules Installer
Running  W32Time            Windows Time
…
```

As you can see, we once again have a list of services whose DisplayNames begin with *windows*; however, this list does not include those services that have the word *audio* anywhere within their names. We did this by specifying first the DisplayName value using the –DisplayName parameter, then specifying the string we want to exclude (*audio*) as the value for the –Exclude parameter.

Positional vs. Named Parameters

Here's a question for you: Let's say you're driving your car and you make a right turn. The road you're turning onto has two lanes going that direction, and you will shortly need to be in the left-hand lane. Do you dutifully turn into the right lane (the lane closest to you), drive for a bit, turn on your left turn signal and carefully change lanes? Or do you simply turn right into the left lane?

Go ahead, you can admit it; you turn directly into the left lane, don't you? While this may or may not be a legal move where you live, in PowerShell it's not only legal but you're encouraged to take as many shortcuts and wide turns as are convenient to you. (And you don't even need to use your turn signal.) Positional parameters are an example of this.

Take a look at this example:

```
Get-Process
```

When you type this cmdlet at the PowerShell command prompt and press Enter you'll get back a list of all the processes running on the local computer. Now let's add a parameter, the process name:

```
Get-Process -Name svchost
```

This will return a list of all processes where the Name of the process is *svchost*:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    324       5     3496       6972    47             864 svchost
    492       8     4364       7908    46             924 svchost
    349      14    57624      42824   133            1036 svchost
    503      14    14292      12768    72            1208 svchost
    757      17    76256      82296   184            1264 svchost
   2571      52   185500     195380   347            1276 svchost
    569      25     7548      13388    79            1468 svchost
   1017      27    16332      20852   117            1720 svchost
    318      25    53524      57428   116            1956 svchost
```

Now let's call Get-Process with a different parameter, this time looking for a process with the ID of 864:

```
Get-Process -Id 864
```

And here's what we get:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    321       5     3448       6912    46             864 svchost
```

You're probably wondering where the shortcuts and wide turns come in. Be patient, we're getting there.

Try this:

```
Get-Process svchost
```

Notice that we left off the –Name parameter, we specified only the value. And what happened? We got the exact same results we got when we *did* included the –Name parameter:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    324       5     3496       6972    47             864 svchost
    492       8     4364       7908    46             924 svchost
    349      14    57624      42824   133            1036 svchost
    503      14    14292      12768    72            1208 svchost
    757      17    76256      82296   184            1264 svchost
   2571      52   185500     195380   347            1276 svchost
    569      25     7548      13388    79            1468 svchost
   1017      27    16332      20852   117            1720 svchost
    318      25    53524      57428   116            1956 svchost
```

The reason we can do this is because the –Name parameter is a positional parameter. Get-Process knows that the first value following the cmdlet name (the value at parameter position 1) is the value for the –Name parameter. You can see how this works by trying the same thing with the ID:

```
Get-Process 864
```

What did you get back? Hey, that's what we got back, too:

```
Get-Process : Cannot find a process with the name "864". Verify the process name and
call the cmdlet again.
```

Get-Process assumes that if a parameter name isn't specified then the first value is the Name. What it's trying to do here is find a process with the name 864, which of course it couldn't do; that's because there is no such process. What all this means is that if you're going to specify an ID, you have to tell Get-Process that you're specifying an ID by including the –Id parameter. There's no other way for Get-Process to know that the value you've put in your command is the ID. The –Id parameter is a named parameter (you must include the parameter name before the value), while the –Name parameter is a positional parameter (the cmdlet can identify the parameter based on where the value is positioned in the command).

Finding Cmdlets

Okay, so we've talked about cmdlets, explained what they are, and even showed you a couple of them. One of the things we noted about cmdlets is that they are the heart and soul of Windows PowerShell. Well, that's great, but how do you find out what cmdlets there are? After all, cmdlets don't do you much good if you don't know about them.

It's time to tell you something about Windows PowerShell, something we haven't mentioned yet: PowerShell has a bit of an ego. Yes, it's true. After spending a little time with PowerShell you'll find that one thing it's very, very good at is telling you all about itself. You can try talking about yourself now and then but PowerShell will largely ignore you. Ask it about itself, however, and you'll get an earful (or at least a screenfull). One example of this is how much PowerShell will tell you about its cmdlets. Do you want to know all of the cmdlets available in Windows PowerShell? Well, there's a cmdlet for that:

```
Get-Command
```

Type Get-Command at the command prompt and you'll get a big, long scrolling list of not only all the cmdlets available but also all of the functions, applications, and various other things. If you want to see only the cmdlets, type this:

```
Get-Command –CommandType cmdlet
```

Once again you'll get a big, long scrolling list (there are over 200 cmdlets, after all), starting something like this:

```
CommandType   Name                  Definition
-----------   ----                  ----------
Cmdlet        Add-Computer          Add-Computer [-DomainName] <String> [-Credential...
Cmdlet        Add-Content           Add-Content [-Path] <String[]> [-Value] <Object...
Cmdlet        Add-History           Add-History [[-InputObject] <PSObject[]>] [-Pas...
Cmdlet        Add-Member            Add-Member [-MemberType] <PSMemberTypes> [-Name...
Cmdlet        Add-PSSnapin          Add-PSSnapin [-Name] <String[]> [-PassThru] [-V...
Cmdlet        Add-Type              Add-Type [-TypeDefinition] <String> [-Language <...
Cmdlet        Checkpoint-Computer   Checkpoint-Computer [-Description] <String> [[-...
```

Here's a little trick that might come in handy: to keep the list from scrolling by too quickly to read, add the pipeline symbol (|) and the word more to the end of your command (we'll explain the pipeline symbol later):

```
Get-Command –CommandType cmdlet | more
```

This command will show you all the cmdlets one screen at a time; simply press the spacebar when you're ready to move on to the next page.

Another thing you can do is narrow down your list a bit. For example, maybe you only want to retrieve the cmdlets that start with the verb New (and yes, in the PowerShell world "New" is a verb). To do this you use the –Verb parameter with the value New:

```
Get-Command -Verb new
```

From that you get back a list something like this:

```
CommandType     Name                    Definition
-----------     ----                    ----------
Cmdlet          New-Alias               New-Alias [-Name] <String> [-Value] <String> ...
Cmdlet          New-Event               New-Event [-SourceIdentifier] <String> [[-Send...
Cmdlet          New-EventLog            New-EventLog [-LogName] <String> [-Source] <S...
Cmdlet          New-Item                New-Item [-Path] <String[]> [-ItemType <Strin...
Cmdlet          New-ItemProperty        New-ItemProperty [-Path] <String[]> [-Name] <...
Cmdlet          New-Module              New-Module [-ScriptBlock] <ScriptBlock> [-Fun...
Cmdlet          New-ModuleManifest      New-ModuleManifest [-Path] <String> -NestedMo...
Cmdlet          New-Object              New-Object [-TypeName] <String> [[-ArgumentLi...
Cmdlet          New-PSDrive             New-PSDrive [-Name] <String> [-PSProvider] <S...
```

Or maybe you want to find all the cmdlets that have to do with the event log. That's easy:

```
Get-Command -Noun eventlog
```

And look what we get back:

```
CommandType     Name                Definition
-----------     ----                ----------
Cmdlet          Clear-EventLog      Clear-EventLog [-LogName] <String[]> [[-Computer...
Cmdlet          Get-EventLog        Get-EventLog [-LogName] <String> [[-InstanceId] ...
Cmdlet          Limit-EventLog      Limit-EventLog [-LogName] <String[]> [-ComputerN...
Cmdlet          New-EventLog        New-EventLog [-LogName] <String> [-Source] <Stri...
Cmdlet          Remove-EventLog     Remove-EventLog [-LogName] <String[]> [[-Compute...
Cmdlet          Show-EventLog       Show-EventLog [[-ComputerName] <String>] [-Verbo...
Cmdlet          Write-EventLog      Write-EventLog [-LogName] <String> [-Source] <St...
```

You can also use wildcards with Get-Command. Let's find all cmdlets with the word Item in them:

```
Get-Command -CommandType cmdlet *item*
```

Notice that we didn't specify a parameter name before the *item* string; that's because the –Name parameter is a positional parameter, which means that Get-Command knows that *item* is the value for the –Name parameter. Once again, take a look at the output:

```
CommandType     Name                   Definition
-----------     ----                   ----------
Cmdlet          Clear-Item             Clear-Item [-Path] <String[]> [-Force] [-Filte...
Cmdlet          Clear-ItemProperty     Clear-ItemProperty [-Path] <String[]> [-Name] ...
Cmdlet          Copy-Item              Copy-Item [-Path] <String[]> [[-Destination] <...
Cmdlet          Copy-ItemProperty      Copy-ItemProperty [-Path] <String[]> [-Destina...
Cmdlet          Get-ChildItem          Get-ChildItem [[-Path] <String[]>] [[-Filter] ...
…
```

Getting Help

This is all making perfect sense so far, right? (Just nod your head and keep reading.) But how are you ever going to know which parameters do what, where they go, or even which parameters are available? How are you going to figure out exactly how to use each cmdlet, or what each cmdlet is supposed to do?

Remember how we told you about PowerShell's ego? Once again, PowerShell is more than happy to tell you all about itself, including just about anything you could want to know about its cmdlets. All you need to do is ask for some help:

```
Get-Help
```

Type Get-Help at the command prompt and press Enter and PowerShell will display help on the screen. As you'll see, typing Get-Help by itself simply gives you some help on the Get-Help cmdlet. If you want help with a different cmdlet, type the cmdlet name in as the first parameter. For example, to find out about the Get-Process cmdlet, type this at the command prompt:

```
Get-Help Get-Process
```

This will give you *some* help. You'll get a description and some syntax statements. But that's not really much help. What you'd really like to see are some examples of how to use the cmdlet. For that you can add the –Examples parameter. Give it a try and see what you get:

```
Get-Help Get-Process –Examples
```

That's more like it. Of course, now all you have is examples, you don't have a full description anymore. And you still don't know what all the parameters are or what they do. If you want *all* the available help on a cmdlet – including examples – use the –Full parameter, like this:

```
Get-Help Get-Process –Full
```

Now, all that is great when you want to know about a specific cmdlet. But what if you want some general PowerShell information? Suppose, for example, you want to know more about how parameters work. (Yes, there is even more than what we've told you. Hard to believe, isn't it?) To get more general information, you can check the "about" help topics. Want to know more about parameters? Type this:

```
Get-Help about_parameters
```

If you want to know which topics are available as about topics, as usual, just ask PowerShell. Use the wildcard character to retrieve a list of all about help topics:

```
Get-Help about*
```

This will retrieve a list which starts out something like this:

```
Name                         Category  Synopsis
----                         --------  --------
about_aliases                HelpFile  Describes how to use alternate names for cmdl...
about_Arithmetic_Operators   HelpFile  Describes the operators that perform arithmet...
about_arrays                 HelpFile  Describes a compact data structure for storin...
about_Assignment_Operators   HelpFile  Describes how to use operators to assign valu...
about_Automatic_Variables    HelpFile  Describes variables that store state informat...
about_Break                  HelpFile  Describes a statement you can use to immediat...
about_command_precedence     HelpFile  Describes how Windows PowerShell determines w...
about_Command_Syntax         HelpFile  Describes the notation used for Windows Power...
```

Getting Around

We've talked a lot about cmdlets (they are, after all, the heart and soul of PowerShell – did we mention that?), but we haven't talked about the PowerShell window itself. For example, how do you navigate your way around the file system? The good news is, if you know how to navigate around the file system in the Cmd.exe window, you know how to navigate around the file system in Windows PowerShell. The better news is that if you know how to navigate around the file system, you also have a pretty good idea how to navigate around the registry. Yes, you read that right, the Windows system registry.

To change from one directory to another, you can use the cd command, like this:

```
PS C:\> cd scripts
```

Press Enter and you'll navigate to the Scripts folder:

```
PS C:\Scripts>
```

Tip: In PowerShell, cd is actually an alias for the Set-Location cmdlet. (Remember, we told you it was all about cmdlets in PowerShell.) So the command **Set-Location scripts** is the same as **cd scripts**. To find out more about aliases, see the **Aliases** section of this Owner's Manual.

You can also display the contents of the current directory with the dir command:

```
PS C:\scripts> dir

    Directory: C:\scripts


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         5/29/2008   8:37 PM            audio
d----          9/6/2007   2:14 PM            begin
-a---         5/20/2008  12:46 PM         26 application.log
-a---        10/31/2007   9:58 PM        155 Audits.xml
-a---          2/5/2008  11:00 AM         54 computers.txt
-a---          5/8/2008   1:27 PM         19 conv.ps1
…
```

Tip: The dir command is an alias for the Get-ChildItem cmdlet. If you're used to working with UNIX, you might want to use the ls command, which is also an alias for Get-ChildItem and, like dir, will retrieve the contents of the current directory.

We mentioned you can do the same thing with the registry. Suppose you want to take a look at the HKEY_CURRENT_USER hive of the registry. You can start by moving out of the file system and into that hive, like this:

```
PS C:\Scripts> cd HKCU:
```

Press Enter and notice the command prompt:

```
PS HKCU:\>
```

Try typing **dir** and you'll see that you really are in the registry. Simply type **C:** and press Enter to return to the file system.

Start Your Engine

Now it’s finally time to get your key out and put it in the ignition. You’ve learned how to start up Windows PowerShell and drive responsibly. The rest of this Owner’s Manual will teach you much more, and trust us, there really is a lot more to learn. For example, we haven’t even mentioned the tailpipe – er, the pipeline – in Windows PowerShell. You’ll *definitely* want to know about that. (See **Piping and the Pipeline** in this Owner’s Manual.) Hey, we haven’t steered you wrong yet, have we?

Piping and the Pipeline

Quick: What one feature truly separates Windows PowerShell from other scripting/shell languages? How many of you said "the pipeline?" Really? That many of you, huh?

It's inevitable: no sooner do you get Windows PowerShell installed then you start hearing about and reading about "piping" and "the pipeline." In turn, that leads to two questions: 1) What's a pipeline?, and 2) Do I even need to *know* about piping and pipelines?

Let's answer the second question first: Do you even need to *know* about piping and pipelines? Yes, you do. There's no doubt that you can work with Windows PowerShell without using pipelines. The question, however, is whether you'd *want* to work with PowerShell without using pipelines. After all, you can order a banana split and ask them to hold the ice cream; that's fine, but at that point you don't really have a banana split, do you? The same is true of the PowerShell pipeline: you can run commands and write scripts without a pipeline. But, at that point, are you really making use of *PowerShell*?

Note. OK, we're exaggerating a little bit: many commands don't *need* a pipeline. In general, however, that's not going to be true of longer and more complicated PowerShell commands and scripts.

Assembling a Pipeline

So then what *is* piping and the pipeline? To begin with, in some ways the term "pipeline" can be considered something of a misnomer. Suppose you have an oil pipeline, like the Alaska Pipeline. You put oil in one end of the pipeline; what do you suppose comes out the other end? You got it: oil. And that's the way a pipeline is supposed to work: it's just a way of moving something, unchanged, from one place to another. But that's not the way the pipeline works in Windows PowerShell.

Note: There's some debate about this. For example, when you "pipe" an object from one part of a command to another, you're simply passing that object – unchanged – from one part of the command to another. There have been some interesting discussions around the PowerShell water cooler about this very topic that involve Kool-Aid and waste management facilities (don't ask), as well as mentions of marketing pipelines and other industry-specific terminology, but we won't confuse you with all that. You're probably confused enough already, and since our goal is to *un*-confuse you, we'll move on.

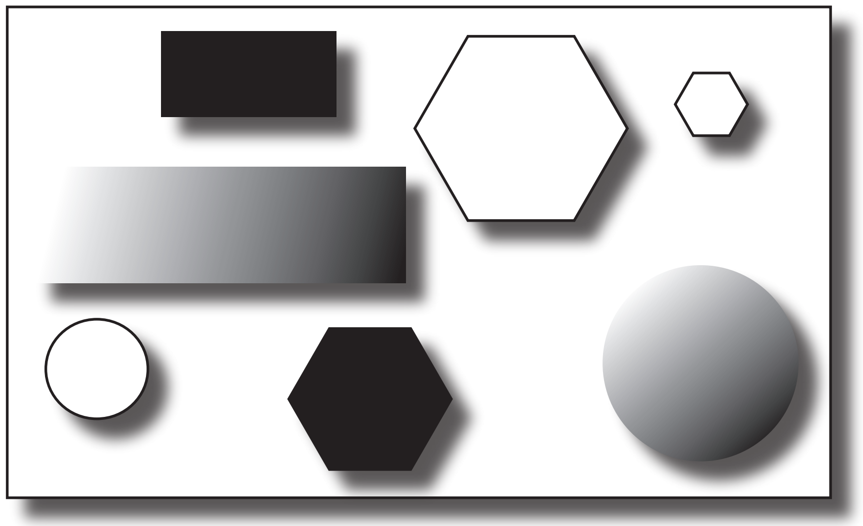
Instead, think of the Windows PowerShell pipeline as being more like an assembly line. With an assembly line you start with a particular thing; for example, you start with a car. However, you don't start with a *finished* car; instead, stations; at each station workers make some sort of modification, welding on doors, adding windows, installing seats. When you're all done you'll have a car; you won't have a tube of toothpaste or a barrel of oil. But thanks to all the changes that were made along the way, you'll have a very different car than the "car" you started with.

Start with a Cmdlet

A similar process takes place when you use the pipeline in Windows PowerShell. For example, suppose there happened to be a cmdlet named Get-Shapes; when you run this cmdlet it returns a collection of all the geometric shapes found on your computer. To call this hypothetical cmdlet you use a command similar to this:

```
Get-Shapes
```

In return, you get back a collection like this one:



The Filtering Station

That's pretty cool – except for one thing. As it turns out, we're only interested in the *clear* shapes. Unfortunately, though, our hypothetical Get-Shapes cmdlet doesn't allow us to filter out items that fail to meet specified criteria. Oh, well; guess we're out of luck, right?

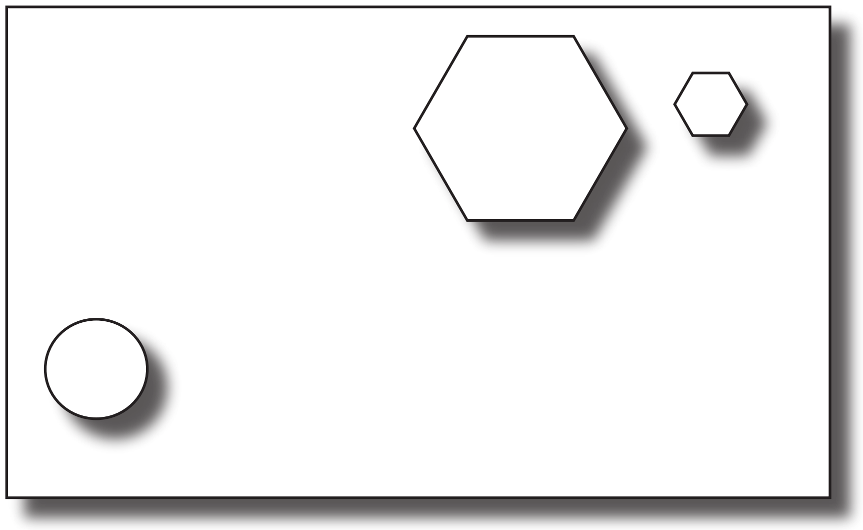
Right.

No, wait, we mean *wrong*. Granted, Get-Shapes doesn't know how to filter out unwanted items. But that's not a problem, because PowerShell's **Where-Object** cmdlet *does* know how to filter out unwanted items. Because of that, all we have to do is use Get-Shapes to retrieve all the shapes, then hand that collection of shapes over to Where-Object and let *it* filter out everything but the clear shapes. In other words:

```
Get-Shapes | Where-Object {$_.Pattern –eq "Clear"}
```

Don't worry about the syntax of the Where-Object cmdlet for now. The important thing to note is the pipe separator character (|) that separates our two commands (Get-Shapes and Where-Object). When we use the pipeline in Windows PowerShell that typically means that we use a cmdlet to retrieve a collection of objects. However, we don't do anything with those objects, at least not right away. Instead, we hand that collection over to a second cmdlet, one that does some further processing (filtering, grouping, sorting, etc.). That's what the pipeline is for.

And in our hypothetical example, the pipeline provides a way for us to filter out everything except the clear shapes:

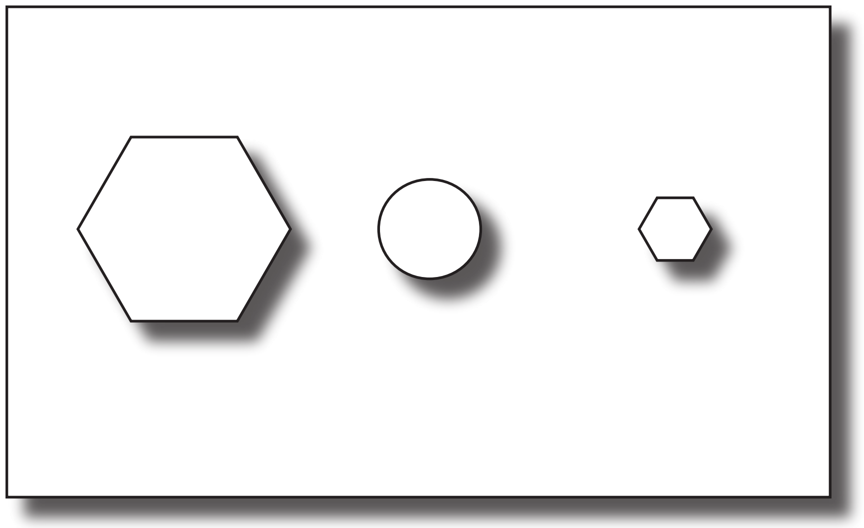


The Sorting Station

That's cool, but what's even cooler is the fact that you aren't limited to just two stations on your assembly line. For example, suppose we want to sort the clear shapes by size. Where-Object doesn't know how to sort things. But Sort-Object does:

```
Get-Shapes | Where-Object {$_.Pattern –eq "Clear"} | Sort-Object Size
```

Does this really work? Of course it does:



A Real-Life Pipeline

Here's a somewhat more practical use of the PowerShell pipeline. The command we're about to show you uses the Get-ChildItem cmdlet to retrieve a list of all the items found in the folder C:\Scripts. The command then hands that collection over to the Where-Object cmdlet; in turn, Where-Object grabs all the items (files and folders) that are greater than 200Kb in size, filtering out all other items. After it finishes filtering, Where-Object hands the remaining items over to the Sort-Object cmdlet, which sorts those items by file size.

The command itself looks like this:

```
Get-ChildItem C:\Scripts | Where-Object {$_.Length -gt 200KB} | Sort-Object Length
```

And when we run the command we get back something along these lines:

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\Scripts
Mode LastWriteTime Length Name
---- ------------- ------ ----
-a--- 2/19/2007 7:42 PM 266240 scores.mdb
-a--- 5/19/2007 9:23 AM 328620 wordlist.txt
-a--- 12/1/2002 3:35 AM 333432 6of12.txt
-a--- 5/18/2007 8:12 AM 708608 test.mdb
```

That's pretty slick, but those of you who've done much scripting seem a little skeptical. "OK, that is nice, but it's not that big of a deal," you say. "After all, if I write a WMI query I can do filtering right in my query. And if I write an ADSI script I can add a filter that limits my collection to, say, user accounts. I'm already doing all this stuff."

Depending on how you want to look at it, that's true; after all, you can use filtering n either a WMI or an ADSI script. However, the approach used when writing a filter in WMI is typically very different from the approach used when writing a filter in ADSI. In turn, an ADSI filter is different from the approach used when writing a filter using the FileSystemObject. The advantage to Windows PowerShell, and to using the pipeline, is that it doesn't matter what kind of data or what kind of object you're working with; you just hand everything off to Where-Object and let Where-Object take care of everything.

Or take sorting, to name another commonly-used operation. If you're doing a database query (including ADO queries against Active Directory) you don't need a pipeline; you can specify sort options as part of the query. But what if you're doing a WQL query against a WMI class? That's a problem: WQL doesn't allow you to specify sort options. If you're a VBScripter that means you have to do something crazy, like write your own sort function or rely on a workaround like disconnected recordsets, just so you can do something as seemingly-simple as sorting data. Is that the case in PowerShell? You already know the answer to that, don't you? Of course that's not the case; in PowerShell you just pipe your data to the Sort-Object cmdlet, sit back, and relax. For example, say you want to retrieve information about the services running on a computer, then sort the returned collection by service status (running, stopped, etc.). Okey-doke:

```
Get-Service | Sort-Object Status | Format-Table
```

Note. You might note that, as a bonus, we took the sorted data and piped it to the Format-Table cmdlet; that means the final onscreen display ends up as a table rather than a list.

Don't Get Carried Away

Yes, this is easy isn't it? In fact, about the only time you'll ever run into a problem is if you get carried away and try pipelining everything. Remember, you can't pipeline something unless it makes sense to use a pipeline. It makes sense to pipeline service information to Sort-Object; after all, Sort-Object can pretty much sort anything. It also makes sense to pipe the sorted information to Format-Table; after all, Format-Table can take pretty much any information and display it as a table.

But consider this command:

```
Sort-Object | Get-Process
```

What's this command going to do? Absolutely nothing. Nor should we expect it to do anything. After all, Sort-Object is designed to sort information and there's nothing here to sort. (Incidentally, that's a hint that Sort-Object should typically appear on the right-hand side of a pipeline. You need to first grab some information and then sort that information.)

Are there exceptions to this rule? Sure. For example, suppose you have a variable $a that contains a collection of data. You can sort that data, and sidestep the pipeline altogether, by using a command like this:

```
Sort-Object –inputobject $a
```

Someday you might actually have to use an approach similar to this; as a beginner, however, you shouldn't worry about it. Instead, you should get into the habit of using piping and the pipeline. Learn the rules first; later on, there will be plenty of time to learn the exceptions.

But even if there was something for Sort-Object to sort this command still wouldn't make much sense. After all, the Get-Process cmdlet is designed to retrieve information about the processes running on a computer; what exactly would Get-Process do with any sorted information handed over the pipeline? For the most part, you first acquire something (a collection, an object, whatever) and then hand that data over the pipeline. The cmdlet on the right-hand side of the pipeline will then proceed to do some additional processing and formatting of the items handed to it.

As we implied earlier, when you do hand data over the pipeline make sure there's a cmdlet waiting for it on the other side. The more you use PowerShell the more you're going to be tempted to do something like this:

```
Get-Process | $a
```

Admittedly, that looks OK – it looks like you want to assign the output of Get-Process to the variable $a then display $a. However, it's not going to work; instead you're going to get an error message similar to this: Expressions are only permitted as the first element of a pipeline.

```
At line:1 char:17
+ Get-Process | $a <<<<
+ CategoryInfo : ParserError: (:) [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : ExpressionsMustBeFirstInPipeline
```

We'll concede that this can be a difficult distinction to make, but pipelines are used to string multiple commands into a single command, with data being passed from one portion of the pipeline to the next. Furthermore, as that data gets passed from one section to another it gets transformed in some way: filtered, sorted, grouped, formatted, whatever. In the invalid command we just showed you, we're not passing any data. We've really got two totally separate commands here: we want to use Get-Process to return information about the processes running on a computer and then, without transforming that data in any way, we want to store the information in variable $a and display that information. Because we really have two independent commands, we need two lines of code:

```
$a = Get-Process
$a
```

And if you're bound and determined to do this all on a single line of code, separate the commands using a semicolon rather than the pipe separator:

```
$a = Get-Process; $a
```

But this isn't pipelining, this is just putting multiple commands on one line.

Bonus Tip

OK, but suppose you wanted to get process information, sort that information by process ID, and then – instead of displaying that information – store the data in a variable named $a. Can you do that? Yes you can, just like this:

```
$a = (Get-Process | Sort-Object ID)
```

What we're doing here is assigning a value to $a. Which value are we assigning it? Well, we're assigning it the value we get back when we call the Get-Process cmdlet and then pipe the returned information to Sort-Object. This command works because we put parentheses around our Get-Process/Sort-Object command. Any time PowerShell parses a command, it carries out the instructions enclosed in parentheses before it does
anything else. In this case, that means PowerShell first gets and sorts process information, then assigns that data to $a. Display the value of $a and see for yourself.

But if you're a beginner, don't worry too much about this bonus example. Get used to using pipelines in the "traditional" way, then come back here and start playing around with parentheses.

More on Pipelining

With any luck, this should be enough to get you started with piping and pipelines. Once you get comfortable with the basic concept you might want a little more technical information about pipelines. If so, just type the following from your Windows PowerShell command prompt:

```
Get-Help about_pipelines
```

Don't assume that you can ignore pipelines and become a true Windows PowerShell user; you can't. You can – and should – ask them to hold the maraschino cherry when ordering a banana split. But if you ask them to hold the ice cream, you'll spend all your time wondering why people make Piping and the such a big deal about banana splits. Don't make that same mistake with Windows PowerShell.

"Don't assume that you can ignore pipelines and become a true Windows PowerShell user; you can't."

Tab Expansion

So you think it's pretty nice that Cmd.exe uses tab expansion for file paths? Just wait until you see what Windows PowerShell does with tab expansion.

One of the big advantages of using Windows PowerShell is that you can just sit at the command prompt and type everything; there's no need to mess around with dialog boxes or mouse clicks or any other GUI nonsense. On the other hand, one of the big *disadvantages* of Windows PowerShell is that you have to sit at the command prompt and type everything: you can't take advantage of dialog boxes or mouse clicks or any other GUI shortcuts.

In other words, if you're the kind of person who likes to type commands from the command prompt then Windows PowerShell is like a dream come true. But what if you don't like to type, or what if you find it difficult to type? In that case, you're just plain out of luck, aren't you?

Well, no, not entirely. Granted, a certain amount of typing will always be required in Windows PowerShell. However, Windows PowerShell also includes a few features that can dramatically reduce the amount of typing required. And one of the coolest of these features is tab expansion.

Tab expansion isn't anything new; in fact, many of you are probably familiar with the tab expansion capabilities built into Cmd.exe. For example, suppose you want to change to the C:\Documents and Settings folder. To do that you need to type this entire command, right?

```
cd "c:\documents and settings"
```

Well, you can if you want to. Alternatively, you can simply type the following and then press the TAB key:

```
cd c:\d
```

If the Documents and Settings folder is the only folder in the root of drive C whose name begins with the letter *D* then you're done: the full folder name will be displayed (including the double quote marks) and you can simply press ENTER and the command will run. But what if you have three or four folders whose names begin with the letter *D*? That's fine: just keep pressing the TAB key and the command shell will dutifully cycle through the complete set of folders whose names begin with the letter *D*. When you finally hit the desired folder press ENTER and let the command shell do your typing for you.

When it comes to file and folder paths Windows PowerShell has this exact same capability. Want to switch to the C:\Documents and Settings folder? All you have to do is type the following and then press the TAB key:

```
cd c:\d
```

If necessary, keep pressing TAB until you see C:\Documents and Settings. At that point press ENTER and – like magic – you'll instantly be transported to the Documents and Settings folder.

Wait, hold your applause; as the saying goes, you ain't seen nothin' yet. Say you want to use the cmdlet Get-AuthenticodeSignature. If you're young and in tip-top physical condition you can type that entire cmdlet name yourself. If you're rich, you can outsource the job and have someone do all that typing for you. Or, if you're lazy (like we are) you can simply type the following and then press the TAB key:

```
get-au
```

That's right: in Windows PowerShell tab expansion not only works with file and folder paths, but it works with cmdlet names as well. Speaking of cmdlets, is that one cmdlet Get-PSSnap-in? Or is it Get-PSSnapin? Or are we way off; maybe it's Get-PowerShellSnapin? To tell you the truth, we don't remember. But that's OK; all we have to do is type the following and then start pressing the TAB key:

```
get-p
```

In no time at all we'll find exactly what we're looking for: Get-PSSnapin.

Tab expansion even works with cmdlet parameters. For example, the Get-Help cmdlet includes a parameter named **-detailed** that returns detailed help about a topic (including examples). You say you like the detailed help but you hate having to type –*detailed*? Then don't. Try this trick to get detailed help about the Get-ChildItem cmdlet:

- Type **get-h** and press TAB.
- Press the spacebar, then type **get-ch** and press TAB.
- Press the spacebar, then type **–** and press TAB. Keep pressing TAB until you see **–Detailed** and then press ENTER.

Pretty slick, huh?

As long as we're on the subject, here's another typing shortcut for you. When it comes to cmdlet parameters you only have to type as much of the parameter name as is needed for Windows PowerShell to know exactly which parameter you're referring to. For example, the Get-Command cmdlet includes the following parameters:

- -CommandType
- -Module
- -Syntax
- -TotalCount
- -Noun
- -Verb
- -Name
- -ArgumentList
- -Debug
- -ErrorAction
- -ErrorVariable
- -WarningAction
- -WarningVariable
- -OutBuffer
- -OutVariable

Need to use the –CommandType parameter? Well, if you want to you can type in the entire parameter name, like so:

```
get-command –commandtype cmdlet
```

On the other hand, because the –CommandType parameter is the only Get-Command parameter whose name begins with the letter *C* you can add this parameter merely by typing **-c**:

```
get-childitem –c cmdlet
```

Nice. By the way, did we mention that tab expansion also works for variable names? Suppose you were silly enough to name a variable $TheMainVariableThatGetsUsedInMyScript. *That's* a lot of typing, especially for a variable name. So then don't type it; just type the following and press TAB any time you need to refer to that variable:

```
$th
```

Cool. Here's one more. Suppose you use the New-Object cmdlet to create an instance of Microsoft Excel:

```
$a = new-object -comobject excel.application
```

Now, type the following, then start pressing the TAB key and see what happens:

```
$a
```

Now you can applaud.

"That's right: in Windows PowerShell tab expansion not only works with file and folder paths, but it works with cmdlet names as well."

Shortcut Keys

Yes, we know: everybody loves to spend their entire day sitting at the keyboard, typing. But, just in case, here are some shortcut keys that can make your Windows PowerShell life much, much easier.

Like all good command shells, Windows PowerShell includes a number of shortcut keys that can lessen the amount of typing needed to get you through a Windows PowerShell session. (Actually, as far as we know even *bad* command shells include shortcuts keys.) This document briefly describes the most commonly-used PowerShell shortcut keys. Because most of these shortcut keys are meaningless without a command history, we'll assume our PowerShell session has been up and running long enough for us to have issued the following set of commands:

1. cd c:\scripts
2. get-childitem -recurse
3. get-executionpolicy
4. set-executionpolicy Unrestricted
5. get-process
6. get-process -name "Notepad"
7. get-history
8. cd c:\windows
9. get-childitem *.dll
10. clear-host

And now let's take a look at the shortcut keys and what they're used for.

Up Arrow
Moves backward through the command history, beginning with the last command typed and working back towards the first command typed.

For example, if the last command you typed was **clear-host** (as in our sample command history) then pressing the Up arrow key one time will display **clear-host** at the command prompt; you can then run this command by pressing ENTER. Pressing the Up arrow key a second time will display the command **get-childitem *.dll**. Pressing the key a third time displays **cd c:\windows**, and so on.

Down Arrow
Moves forward through the command history. Continuing with the Up arrow example, suppose you backed your way through the command history (starting with the latest command and moving backwards towards the first command) until **get-process –name "Notepad"** is displayed at the command prompt. If you now press the Down arrow key you'll move forward through the command history and **get-history** will be displayed. Press the Down arrow key a second time and **cd c:\windows** will be displayed. Etc., etc.

PgUp
Displays the first command in the command history. In our example, that causes **cd c:\scripts** to be displayed at the command prompt.

PgDn
Displays the last command in the command history. In our example, that's the **clear-host** command.

Left Arrow
Moves the cursor one character to the left on the command line. For example, if the cursor is under the\ in command **cd c:\scripts**, pressing the Left arrow key will move the cursor under the **:**.

Right Arrow
Moves the cursor one character to the right on the command line. For example, if the cursor is under the **:** in **cd c:\scripts**, pressing the Right arrow key will move the cursor under the ****.

Home
Moves the cursor to the beginning of the command line.

End
Moves the cursor to the end of the command line.

Control + Left Arrow
Moves the cursor one "word" to the left on the command line. For example, suppose the command **get-process –name "Notepad"** is displayed at the command line; in addition, suppose that the cursor is under any of the characters in the "word" get-process. (In this case, "words" are strings of characters delineated by blank spaces.) Holding down the Ctrl key and pressing the Left arrow key will cause the cursor to move beneath the **g** in **get-process**.

Control + Right Arrow
Move the cursor one "word" to the right on the command line. For example, suppose the command **get-process –name "Notepad"** is displayed at the command line; in addition, suppose that the cursor is under any of the characters in the "word" **–name**. (In this case, "words" are strings of characters delineated by blank spaces.) Holding down the Ctrl key and pressing the Right arrow key will cause the cursor to move beneath the **"** in **"Notepad"**.

Control + c
Cancels the current command. If you are partway through typing a command and then press Ctrl+c Windows PowerShell will ignore everything you've typed on the line and present you with a new "blank" command line.

F2
Creates a new command line from the partial contents of your last command line. For example, suppose your previous command was **get-process –name "Notepad"**. If you press F2, PowerShell will respond with the following request:

```
Enter char to copy up to:
```

If you type **“** (representing the first quotation mark) PowerShell will insert the text from your previous command, up to (but not including) the double quote mark:

```
get-process –name
```

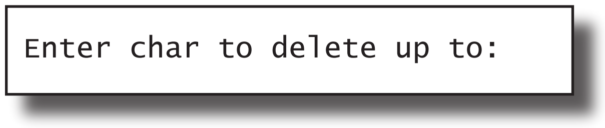
Note that PowerShell will always go to the first instance of a character. If you type an **e** in the dialog box PowerShell will begin “typing” until it encounters the first *e* in *get-process*. Therefore, all that will be displayed on your screen is the following:

```
g
```

To cancel when asked to enter a character, press Enter without entering anything.

F3
Displays your previous command. This is equivalent to pressing the Up arrow key once. However, while you can press Up arrow multiple times to continue cycling through your command history, pressing F3 additional times has no effect: only your last command will be displayed.

F4
Beginning from the current cursor position, F4 deletes characters up to the specified character. For example, suppose the command **get-process –name “Notepad”** is displayed, and the cursor is beneath the *c* in *process*. If you press F4, the following dialog box appears:



```
Enter char to delete up to:
```

If you press **–** PowerShell will begin deleting characters from the command prompt and continue deleting characters until it encounters a **–** character. In this case, that means the following (nonsensical) command will be displayed:

```
get-pro-name “Notepad”
```

If you enter a character (say, *z*) which does not appear in the command string then all the remaining characters in the string will be deleted:

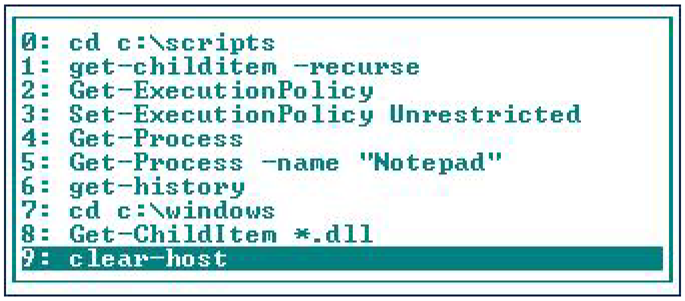
```
get-pro
```

This also occurs if you simply press ENTER when the dialog box appears.

To cancel this operation, press Ctrl+Z.

F5
Like the Up arrow key, F5 moves you backward through the command history.

F7
Displays a "dialog box" that allows you to select a command from your command history:



```
0: cd c:\scripts
1: get-childitem -recurse
2: Get-ExecutionPolicy
3: Set-ExecutionPolicy Unrestricted
4: Get-Process
5: Get-Process -name "Notepad"
6: get-history
7: cd c:\windows
8: Get-ChildItem *.dll
9: clear-host
```

Scroll through the dialog box using the Up and Down arrow keys; when you find a command you want to execute press ENTER. Alternatively, locate the desired command and then press the right arrow key; that will "type" the command at the command prompt but will not execute that command. To execute the command, you must press ENTER.

To dismiss the dialog box without selecting a command, press Escape.

F8
Moves backwards through the command history, but only displays commands matching text you type at the command prompt. For example, suppose you type **set** and then press F8. In that case (and using our sample command history) PowerShell will only display the command **set-executionpolicy**. Why? Because that's the only command in the history that matches the string **set**.

Now, type **cd** and press F8. This time around, PowerShell will display the command **cd c:\windows**. Press F8 a second time and the command **cd c:\scripts** will be displayed. That's because, in this case, we have two commands that match the string **cd**.

F9
Enables you to run a specific command from the command history. When you press F9 PowerShell prompts you to command number, corresponding to the numbers displayed in the history "dialog box" (see **F7**):

```
Enter command number:
```

To run a command, type the appropriate number and press ENTER. Using our sample history, typing 4 and pressing ENTER runs the command **get-process**.

To cancel this operation, press Escape.

Customizing the Console

When it came to automobiles, Henry Ford once said that customers could have "any color they want. As long as it's black." That's why Henry Ford wasn't allowed to design the console window for Windows PowerShell.

Many of you will find this hard to believe, but there was a time – long, long ago – when people would get things and then use those items *exactly as they got them*! That's right: no customization, no "tricking out," no modding, no skinning, no nothing. You just took the thing out of the box and used it the way nature – and the manufacturer – intended.

Yes, we know: barbaric.

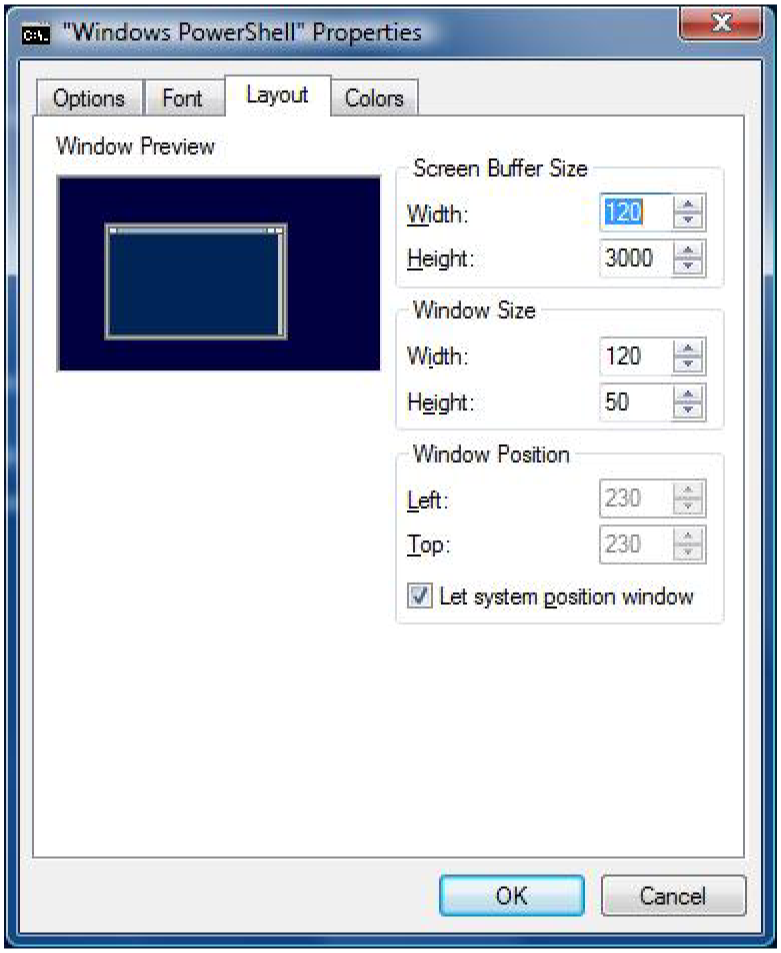
Today, of course, things are very different. No one would ever *dream* of using an off-the-shelf item in this day and age; instead, everything needs to be personalized. And that might have some of you a bit leery about trying Windows PowerShell. After all, if you can't customize and trick out this new software, well, then what's the point in even using it in the first place?

If you happen to be one of those people, we have good news for you: we're not sure any of this qualifies as "tricking out," but you *can* custom-tailor Windows PowerShell to a surprising degree. Don't like commands named Get-ExecutionPolicy and Set-AuthenticodeSignature? Fine; just create a new alias and give these commands any name you want to give them. (See the **Aliases** section of this Owner's Manual.) Don't like the default format PowerShell uses when displaying information for a particular type of object? Fine; create your own .PS1XML file and change the default format. Don't like the way the Windows PowerShell console looks? Fine; change the console size, change the console fonts or colors, change pretty much whatever you want. It's entirely up to you. And, as it turns out, modifying the look of the console window is remarkably easy.

As we all know, Windows PowerShell is practically brand-new, cutting-edge technology; nevertheless, this new functionality is hosted within the same console window that William Shakespeare used when writing *Hamlet*. (That is, the same console window used by Cmd.exe.) We're the first to admit that there are some definite disadvantages to this. For one thing, don't bother trying to use Ctrl+C and Ctrl+V to copy and paste text inside the console window; it's not going to work. However, there is at least one *advantage* to reusing the same shell as Cmd.exe: if you know how to modify the Cmd.exe console then you already know how to modify the Windows PowerShell console.

Point taken: what if you *don't* know how to modify the Cmd.exe console? That's OK; in just a second we'll show you how to use the GUI to modify the console window. And then, just for the heck of it, we'll show you some programmatic ways to modify console properties. You say you don't like the way your PowerShell window looks? Then read on.

Let's kick things off by showing you how to use the GUI to modify the console window. (As a bonus, you can use this approach to modify Cmd.exe as well as PowerShell.) To begin with, start Windows PowerShell. (Always a good place to start.) When the PowerShell window appears, click the icon in the upper left-hand corner and the click **Properties**. That will bring up a dialog box that looks like this:



Screen Buffer Size
Width: 120
Height: 3000

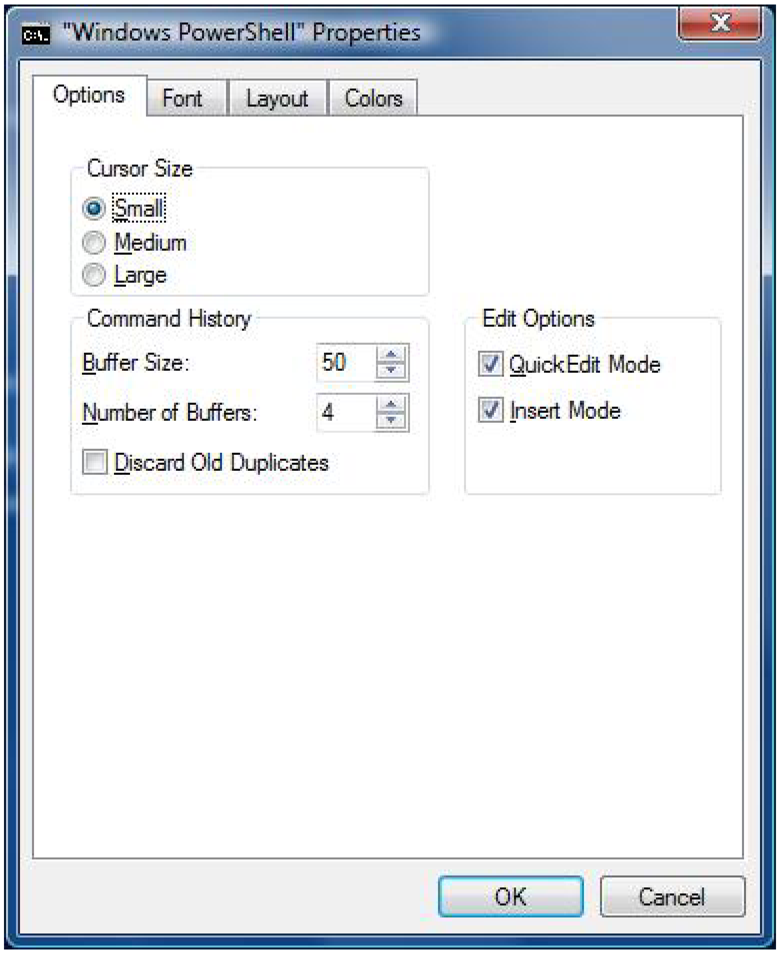
Window Size
Width: 120
Height: 50

Window Position
Left: 230
Top: 230
[✓] Let system position window

Recognize that? From here you can start clicking the various tabs – and start modifying the console window – to your heart's content.

"Many of you will find this hard to believe, but there was a time – long, long ago – when people would get things and then use those items *exactly as they got them*! "

We're not going to tell you how big or how colorful you should make your console window; that's up to you. However, we *will* recommend that you click on the **Options** tab and select both **QuickEdit Mode** and **Insert Mode**:



Cursor Size
[✓ Small]
Medium
Large

Command History
Buffer Size: 50
Number of Buffers: 4
☐ Discard Old Duplicates

Edit Options
✓ QuickEdit Mode
✓ Insert Mode

Why? Well, QuickEdit Mode enables you to use the mouse to copy and paste within the PowerShell window. Assuming this mode is enabled, you can copy something to the Clipboard by highlighting the text with the mouse and then pressing ENTER; to paste text from the Clipboard, position the cursor and then click the right-mouse button. Insert Mode, meanwhile, enables you to insert text when typing. If Insert Mode is disabled then new text that you type in will overwrite any existing text. (Give it a try and you'll see what we mean.)

Tip. Here's something cool. Suppose you have a PowerShell script, either a simple one that looks like this or a more complicated example:

```
cd C:\Scripts
Get-ChildItem -recurse
```

Now, suppose you're looking at this script in Notepad or on the Web. Highlight both lines, copy them, switch over to PowerShell and paste in the script. PowerShell will go ahead and run all the commands: it will change the working folder to C:\Scripts and then run the Get-ChildItem cmdlet. The point of all that? You can paste in an entire script and PowerShell will dutifully execute each line in turn; you do *not* have to paste a script in line-by-line.

When you're finished making all your changes click **OK**.

Granted, it's not quite modding or skinning. But at least it's something, right?

Using a Script to Modify Console Properties

Now, it's great that we can start PowerShell and then use a series of dialog boxes to modify the console window. However, we're script writers: we hate to do *anything* by hand if there's a way to do it using a script. So here's the question: can we modify properties of the console window programmatically?

As it turns out, there are several properties of the console window that are easy to modify using a script. (You can modify some of the not-so-easy to modify properties as well, but because that involves making changes to the registry we'll skip those for now.) In particular, you can easily change the console window colors (both the color of the window itself and the color of the text); the console window title; and the size of the window.

Note. So why would you *want* to programmatically change the properties of the console window? Well, the truth is, maybe you don't. On the other hand, suppose you have multiple PowerShell sessions (or product-specific management shells) running at the same time. (Something you not only *can* do but sooner or later *will* do.) Changing the window title and/or changing the window colors makes it much easier for you to differentiate between Session A, Session B, and Session C. Likewise, modifying the window size helps ensure that your data will fit just the way you need it to.

The secret to programmatically modifying console properties is to use the **Get-Host** cmdlet. Typically, Get-Host is used to display information about PowerShell itself, particularly the version number and regional information. For example, type **Get-Host** from the PowerShell command prompt and you'll get back information similar to this:

```
Name             : ConsoleHost
Version          : 2.0
InstanceId       : 89c72fa7-c7f0-4766-8615-451990b15f70
UI               : System.Management.Automation.Internal.Host.
InternalHostUserInterface
CurrentCulture   : en-US
CurrentUICulture : en-US
PrivateData      : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
IsUnspacePushed  : False
Runspace         : System.Management.Automation.Runspaces.LocalRunspace
```

So what's the big deal here? The big deal is the **UI** property, which is actually a portal to a child object (which, as you can see from the output above, is derived from the .NET Framework class System.Management.Automation.Internal.Host.InternalHostUserInterface). The UI object, in turn, has a property named **RawUI**, which gives us access to console properties such as window colors and title. Take a look at what we get back when we run the command **(Get-Host).UI.RawUI**:

```
ForegroundColor       : DarkYellow
BackgroundColor       : DarkMagenta
CursorPosition        : 0,125
WindowPosition        : 0,76
CursorSize            : 25
BufferSize            : 120,3000
WindowSize            : 120,50
MaxWindowSize         : 120,82
MaxPhysicalWindowSize : 175,82
KeyAvailable          : False
WindowTitle           : Windows PowerShell
```

Many of these properties can be changed using a script. *How* can they be changed using a script? We were just about to show you.

But first, another question: Why the parentheses in the command **(Get-Host).UI.RawUI**? Well, any time Windows PowerShell parses a command, it performs operations in parentheses before it does anything else. In this case, that means that PowerShell is going to first run the Get-Host cmdlet and then, after that command has completed, access the UI property of the returned object and then access the RawUI property of the UI object (which happens to be yet another object). The single line of code **(Get-Host).UI.RawUI** is shorthand for the following:

```
$a = Get-Host
$b = $a.UI
$c = $b.RawUI
```

But who cares about the technical details, right? Let's do something fun, like change the background color and the foreground (text) color of the console window. Here's a three-line script that gives us yellow text on a green background:

```
$a = (Get-Host).UI.RawUI
$a.BackgroundColor = "green"
$a.ForegroundColor = "yellow"
```

Note. After running the script you might want to call the **Clear-Host** function (or its alias, **cls**) in order to "refresh" the screen. Otherwise only part of the window will feature yellow text on a red background.

As you can see, there isn't much to the script. We create an object reference (named $a) to the UI.RawUI object, and then we simply assign new values to the **BackgroundColor** and **ForegroundColor** properties.

Speaking of which, here are the different colors available to you:

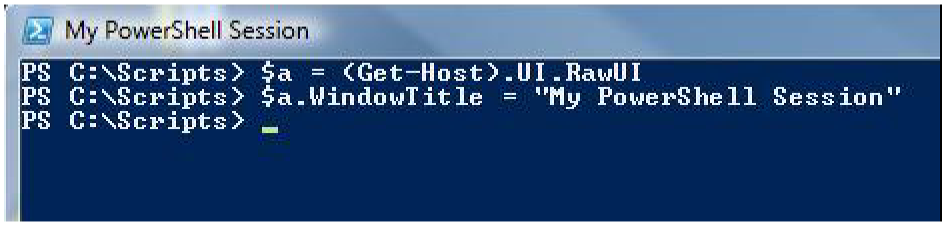
Black	Blue
DarkBlue	Green
DarkGreen	Cyan
DarkCyan	Red
DarkMagenta	Magenta
DarkYellow	Yellow
Gray	White
DarkGray	DarkRed

Give it a try and see what happens. And yes, it *is* a good idea to wear eye protection if you choose this color scheme.

That was pretty cool, wasn't it? Now, let's see if we can change the window title:

```
$a = (Get-Host).UI.RawUI
$a.WindowTitle = "My PowerShell Session"
```

Wow; that took just *two* lines of code. Again we create an object reference to the UI.RawUI object. And then this time we assign a new value to the **WindowTitle** property. The net result? This:



```
PS C:\Scripts> $a = (Get-Host).UI.RawUI
PS C:\Scripts> $a.WindowTitle = "My PowerShell Session"
PS C:\Scripts> _
```

Let's make one more little change before we call it a day: let's change the size of the console window itself. First let's take a look at the code, then we'll explain how it all works:

```
$a = (Get-Host).UI.RawUI
$b = $a.WindowSize
$b.Width = 40
$b.Height = 10
$a.WindowSize = $b
```

This is a tiny bit more complicated, simply because the **WindowSize** property is yet another object. In our first line of code we once again create a reference to the UI.RawUI object. We then grab the WindowSize property of that object and assign it to a new object ($b). That's what this line of code is for:

```
$b = $a.WindowSize
```

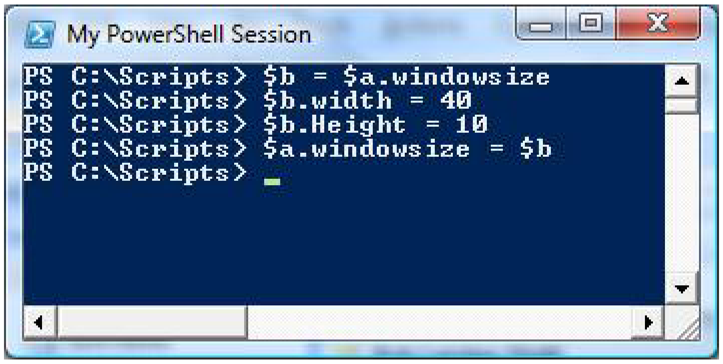
That gives us access to all the properties of WindowSize. At this point we can go ahead and assign new values to two of these properties, **Width** and **Height,** to $b. That's what we do here:

```
$b.Width = 40
$b.Height = 10
```

And then once we have $b's properties configured we can go ahead and assign $b (and its values) to the WindowSize property of $a:

```
$a.WindowSize = $b
```

Granted, it's not the most intuitive thing in the world. But it works:



```
PS C:\Scripts> $b = $a.windowsize
PS C:\Scripts> $b.width = 40
PS C:\Scripts> $b.Height = 10
PS C:\Scripts> $a.windowsize = $b
PS C:\Scripts> _
```

Making the Changes (Semi) Permanent

So let's assume you create the perfect console window: it's got the right colors and the right title, and the window is sized perfectly. How can you make sure that you get these same settings each and every time you run Windows PowerShell?

Here's one way: put the appropriate commands into your PowerShell profile. To find out how to do that, take a look at the section **Windows PowerShell Profiles** in this Owner's Manual. Once you've put all the commands you need into your profile, every time Windows PowerShell starts up your console will look exactly the way you specified.

Windows PowerShell Aliases

*So, would you rather type something like **Get-AuthenticodeSignature** or would you rather type something like **gas**? We had a feeling you'd say that.*

Samuel Clemens. Archibald Leach. Ramón Gerardo Antonio Estévez. William Sydney Porter. The list goes on…. And exactly what is this a list of? Well, as you might have figured out it's a list of given names of people who decided they didn't want to (or couldn't) become famous with those names. Instead they took on new names, or aliases, to be more easily identified and remembered. As it turns out, Windows PowerShell has decided to join the other celebrities in utilizing aliases. How so? Read on.

It's true that Windows PowerShell, before it became Windows PowerShell, began life with a given name of Monad. It then briefly tried out MSH (pronounced "mish"), and finally settled on the catchy, rolls-smoothly-off-the-tongue, Windows PowerShell. Given that lifecycle you might think of Windows PowerShell as an alias. The problem is that if we call it that, the Microsoft lawyers will be very unhappy: it was a legal name change. Since we find it best not to make the lawyers unhappy, we won't use the term "alias" to refer to Windows PowerShell. But we *will* use the term "alias" to talk about a certain renaming feature *within* PowerShell. These aliases serve the same purpose as the aliases we mentioned at the beginning of this article: they make certain things in Windows PowerShell easier to identify and remember. (Although in some cases, if you're not careful, an alias can be pretty effective at *hiding* an identity.)

Aliases Defined

Within PowerShell, an alias is another name assigned to a cmdlet, function, script, executable, and so on. Just about anything you can run from the PowerShell command prompt can have an alias assigned to it. We'll explain how this works and why you'd even care about this by first showing you some of the aliases built-in to PowerShell. After that we'll show you how to make aliases of your own. And then, after we've covered that, we'll show you how to make sure the aliases you create stick around from one PowerShell session to another.

Are you ready for all that? Don't worry, this will be easy. Even easier than remembering who Norma Jeane Mortenson was.

Built-In Aliases

Windows PowerShell 2.0 ships with well over 100 built-in aliases. You can find these aliases with the Get-Alias cmdlet:

```
PS C:\scripts> get-alias

CommandType     Name          Definition
-----------     ----          ----------
Alias           %             ForEach-Object
Alias           ?             Where-Object
Alias           ac            Add-Content
Alias           asnp          Add-PSSnapIn
Alias           cat           Get-Content
Alias           cd            Set-Location
…
```

All the built-in aliases are simply short versions of cmdlet names. For example, the alias for Get-Alias is gal:

```
PS C:\scripts> gal

CommandType     Name          Definition
-----------     ----          ----------
Alias           %             ForEach-Object
Alias           ?             Where-Object
Alias           ac            Add-Content
Alias           asnp          Add-PSSnapIn
Alias           cat           Get-Content
Alias           cd            Set-Location
Alias           chdir         Set-Location
Alias           clc           Clear-Content
Alias           clear         Clear-Host
…
```

You can use these aliases alone, like we just did, or anywhere within a command string where you'd use the cmdlet. For example, here's a command that retrieves the cmdlet associated with the alias ac:

```
PS C:\scripts> gal -name ac

CommandType     Name          Definition
-----------     ----          ----------
Alias           ac            Add-Content
```

You can imagine how much typing this can save you in the long run. Just to demonstrate, here's a command that retrieves the five smallest text files in the C:\ folder:

```
get-childitem C:\*.txt | sort-object -property length | select-object -last 5
```

Here's that same command using built-in aliases:

```
ls C:\*.txt | sort -property length | select -last 5
```

That's better.

Note. If you use shorthand for the parameters you can compact this even more:

ls C:\*.txt | sort -p length | select -l 5

But keep in mind that parameter resolution is a completely different feature than aliases. We're not going to talk about parameters here. If this just confused you, ignore this note and keep reading.

A lot of the aliases are built-in simply to give you a quicker way to access cmdlets. However, some of them are there to make sure you can do things such as navigate your way around the file system in the command window by using familiar commands. For example, here are the commands you can use to list the items in the current directory:

Shell	Command
MS-DOS	Dir
Unix	Ls
Windows PowerShell	Get-ChildItem

Admittedly, PowerShell's commands sometime seem a little cumbersome compared to the others. But with built-in aliases you can use any of these commands in PowerShell (in addition to the alias gci) to retrieve the contents of a folder.

Finding Aliases

We already showed you how to get a list of all the aliases available (the Get-Alias cmdlet). Here's how to get a list of aliases sorted by alias:

```
get-alias | sort-object
```

Want to sort by definition (the cmdlet, function, etc. that the alias is an alias for)? Okay:

```
get-alias | sort-object definition
```

How about this: let's retrieve the definition associated with a given alias. We'll try the cd alias:

```
PS C:\scripts> get-alias cd

CommandType     Name          Definition
-----------     ----          ----------
Alias           cd            Set-Location
```

This is all pretty simple, right? The next logical step would be to find all the aliases associated with a given cmdlet (or definition). That's pretty easy too:

```
PS C:\scripts> Get-Alias -Definition Set-Location

CommandType     Name          Definition
-----------     ----          ----------
Alias           cd            Set-Location
Alias           chdir         Set-Location
Alias           sl            Set-Location
```

Creating Aliases

Okay, this is all very nice, but what if you don't like the built-in aliases? Or what if you want to set an alias for a function that you've created? As it turns out, this is no problem at all. Simply use the Set-Alias cmdlet. Suppose you want to retrieve the current date. You could do that with the Get-Date cmdlet:

```
PS C:\scripts> Get-Date

Wednesday, September 16, 2009 3:47:56 PM
```

That's simple enough, but let's make it even simpler – let's set an alias for Get-Date:

```
Set-Alias d Get-Date
```

Logically enough, we use the Set-Alias cmdlet to create an alias. We pass two parameters to this cmdlet: the new alias (d) and the cmdlet being aliased (Get-Date). Let's try it out:

```
PS C:\scripts> d

September 16, 2009 3:49:51 PM
```

The cmdlet works exactly the same, we just use an alias to reference it. For example, you can pass parameters to the alias in the same way you'd pass them to the cmdlet:

```
PS C:\scripts> d -displayHint date

September 16, 2009
```

As you can see, we used the displayHint parameter of the Get-Date cmdlet to display only the date portion of the date/time value, but we used that parameter with the d alias.

You can also alias functions. Here's a function that finds the default printer on the local computer.:

```
function FindDefaultPrinter
{
    Get-WMIObject –query "Select * From Win32_Printer Where Default = TRUE"
}
```

To run this function, simply type the name of the function, FindDefaultPrinter, at the command prompt:

```
PS C:\scripts> FindDefaultPrinter

Location      : USA/REDMOND, WA/42/FLOOR4/4032
Name          : \\PRINTER1\HP LaserJet
PrinterState  : 0
PrinterStatus : 3
ShareName     : HP LaserJet
SystemName    : \\PRINTER1
```

What's that? FindDefaultPrinter is a big long name you don't want to type in every time you want to check for the default printer? Okay, how about this:

```
PS C:\scripts> dp

Location      : USA/REDMOND, WA/42/FLOOR4/4032
Name          : \\PRINTER1\HP LaserJet
PrinterState  : 0
PrinterStatus : 3
ShareName     : HP LaserJet
SystemName    : \\PRINTER1
```

Will that work? Well, no, not unless you do this:

```
Set-Alias dp FindDefaultPrinter
```

Now it will work.

You can also alias executables. Want to start Microsoft Excel from the command prompt? Why would you ever want to do that? Oh well, that's none of our business. But if you do, you could do it like this (assuming you have the default installation for Excel 2007):

```
."C:\Program Files\Microsoft Office\Office12\Excel.exe"
```

Note. The dot (.) is required. If you leave it out, PowerShell will just display the path that's within the quotes. Not exactly what we had in mind.

Or you could simply alias the command:

```
Set-Alias excel "C:\Program Files\Microsoft Office\Office12\Excel.exe"
```

Now all you have to type is this:

```
excel
```

It doesn't get much easier than that.

Keeping Aliases Around

You've now spent a significant amount of time setting up all your aliases. Okay, maybe "significant" is exaggerating a little, but nevertheless you did spend *some* time on this. And all system administrators know exactly how valuable time is and how little we have to waste, which is why we set up aliases in the first place. Now suppose you shut down Windows PowerShell, move on to some other tasks, and later restart PowerShell. Hmmm, let's use our alias **d** to see what day it is today:

```
The term 'd' is not recognized as a cmdlet, function, operable program, or script file.
Verify the term and try again.
At line:1 char:2
+ d <<<<
    + CategoryInfo          : ObjectNotFound: (d:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

Wait a minute – didn't we just set the alias d to represent the cmdlet Get-Date? Yes, as a matter of fact we did. But then we shut down Windows PowerShell. *Big* mistake. Don't ever, ever, shut down Windows PowerShell.

Well, not unless you save your aliases first.

There are two ways to do this. The first approach is to export your aliases to a file. This is much easier than it might sound. That's because PowerShell makes it easy by providing the Export-Alias cmdlet:

```
Export-Alias MyAliases.csv
```

You need to pass Export-Alias only one parameter (although there are more parameters, which we'll talk about in a moment): the name of the text file that you want to save your aliases to. By default the file is saved as a comma-separated values file (which is why we gave it a .csv file extension, although you could also simply give it a .txt file extension).

Note: If you don't want the file saved to the current path, you must specify the full path to the file:

```
Export-Alias C:\Files\MyAliases.csv
```

This command will export all aliases – user-defined and built-in – to the given file. Keep that in mind, because that could be a bit of a nuisance. We'll show you that in a moment, too. But first, let's talk about what we need to do with these aliases now that we've exported them.

As we mentioned, when you close your PowerShell session all the aliases you defined will disappear. That means that the next time you open PowerShell, you'll need to redefine all your aliases. However, if you exported them to a file first, you can simply import them back into your session and all your aliases will be back. You do that with the Import-Alias cmdlet:

```
Import-Alias MyAliases.csv
```

Simply pass the name of the file that contains your exported aliases (including the full path if the file isn't in the current directory) to the Import-Alias cmdlet. Now you can use any of the aliases you had defined before the import.

What's that? You tried this and got a big long list of errors? A list like this:

```
Import-Alias : Alias not allowed because an alias with the name 'ac' already exists.
At line:1 char:13
+ Import-Alias  <<<< MyAliases.csv
Import-Alias : Alias not allowed because an alias with the name 'asnp' already exists.
At line:1 char:13
+ Import-Alias  <<<< MyAliases.csv
Import-Alias : Alias not allowed because an alias with the name 'clc' already exists.
At line:1 char:13
+ Import-Alias  <<<< MyAliases.csv
…
```

First of all, don't worry: your aliases are still there, and if you try one you'll see that the import succeeded and your aliases will work just fine. So why all the errors? Remember when we said Export-Alias will export *all* aliases, user-defined and built-in? Well, the user-defined aliases were wiped out when you closed PowerShell, but the built-in aliases weren't. That means that, when you run Import-Alias, all those built-in aliases that were exported will be re-imported. And that's a problem, because you can't overwrite existing aliases. Make sense? On the one hand these errors don't hurt anything, they simply inform you that you can't import a built-in alias because it already exists. On the other hand, who wants to see all those error messages every time they import the file? Not us, that's for sure. But don't worry, there's something you can do about it.

One simple thing you can do is edit the alias file after you export it. Your exported file will look something like this:

```
# Alias File
# Exported by : kenmyer
# Date/Time : Wednesday, September 16, 2009 6:58:03 PM
# Machine : ATL-FS-01
"ac","Add-Content","","ReadOnly, AllScope"
"asnp","Add-PSSnapIn","","ReadOnly, AllScope"
"clc","Clear-Content","","ReadOnly, AllScope"
"cli","Clear-Item","","ReadOnly, AllScope"
"clp","Clear-ItemProperty","","ReadOnly, AllScope"
"clv","Clear-Variable","","ReadOnly, AllScope"
```

Simply delete all the references to built-in aliases. Don't worry, this isn't hard to do: all the built-in aliases are listed first; your custom aliases are all at the bottom of the file:

```
…
"set","Set-Variable","","AllScope"
"type","Get-Content","","AllScope"
"d","get-date","","None"
"dp","FindDefaultPrinter","","None"
"excel","C:\Program Files\Microsoft Office\Office12\Excel.exe","","None"
```

See those last three, *d*, *dp*, and *excel*? Those are the aliases we added earlier. They'll always be at the bottom.

The other thing you can do is add specific aliases to the file rather than doing a global export. You do this by passing a second parameter to Export-Alias, the name of the alias you want to export:

```
Export-Alias MyAliases.csv d
```

Here we've passed the alias d, which we set up for Get-Date. Here's what our alias file looks like now:

```
# Alias File
# Exported by : kenmyer
# Date/Time : Wednesday, September 16, 2009 6:58:03 PM
# Machine : ATL-FS-01
"d","get-date","","None"
```

Here's something important to know: every time you call Export-Alias, the current file is overwritten. That means that if you now call Export-Alias to export a different alias, d will be gone and will be replaced by the new alias. Like this:

```
Export-Alias MyAliases.csv dp

# Alias File
# Exported by : kenmyer
# Date/Time : Wednesday, September 16, 2009 6:58:03 PM
# Machine : ATL-FS-01
"dp","FindDefaultPrinter","","None"
```

But wait, here's something even *more* important to know: you can pass another parameter to Export-Alias to make sure the new aliases are appended rather than overwritten:

```
Export-Alias MyAliases.csv d –append
```

Here's what we have now:

```
# Alias File
# Exported by : kenmyer
# Date/Time : Wednesday, September 16, 2009 6:58:03 PM
# Machine : ATL-FS-01
"dp","FindDefaultPrinter","","None"
# Alias File
# Exported by : kenmyer
# Date/Time : Wednesday, September 16, 2009 6:58:03 PM
# Machine : ATL-FS-01
"d","get-date","","None"
```

Granted, this file could still use some cleanup, but you don't *have* to clean it up. (Although, if you want to, it's easy enough to do.) Either way, you won't get any more error messages.

Note that exporting and importing aliases isn't just handy for keeping aliases from one session of PowerShell to the next; you can also copy the exported file to different machines and import them into the instance of PowerShell on those machines. Nice, huh?

We mentioned that, by default, exported aliases are saved to a comma-separated values (csv) file. That's nice, but suppose, instead of repeatedly importing aliases, you want to run a script that will run the Set-Alias commands to re-create your aliases? Well, you're in luck, because the Export-Alias cmdlet has a parameter named –As that lets you specify the output as a script file:

```
Export-Alias -As script MyAliases.ps1 d
```

Here we've called the Export-Alias cmdlet with the –As parameter, followed by the value "script". We then pass the name of the file we're exporting to, this time giving it a .ps1 extension (the file extension for PowerShell scripts), and the alias (d) that we want to export. When we open the MyAliases.ps1 file, this is what we see:

```
# Alias File
# Exported by : kenmyer
# Date/Time : Wednesday, September 16, 2009 7:21:50 PM
# Machine : ATL-FS-01
set-alias -Name:"d" -Value:"get-date" -Description:"" -Option:"None"
```

Notice that the command isn't exactly the same as what we typed to the command line to set the alias. Export-Alias doesn't look back into your history to see what aliases were set, it simply looks at the available aliases (or the specified aliases, such as, in this case, d) and creates a Set-Alias command that would re-create that alias.

The next time you close Windows PowerShell and then reopen it, you can type this at the command prompt:

```
PS C:\scripts> . .\MyAliases.ps1
```

This runs the script, which includes the Set-Alias statement. Now when you type d at the command prompt, you get the output from Get-Date, the cmdlet that our alias refers to.

Note. The extra dot (.) in front of the path isn't a mistake. If you don't include that, the Set-Alias statement will run, but the results won't be saved to the current instance of Windows PowerShell. Confused? Read the part about running scripts in the "Introduction to Scripting" section of this Owner's Manual.

Now, at first glance this might seem like more trouble – or at least not any simpler – than saving aliases to a .csv file and running the Import-Alias cmdlet. However, there's one more thing you can do to keep your aliases from one PowerShell session to another. Simply add the Set-Alias commands (the ones that you generated using the –As Script parameter) to your Windows PowerShell profile and your aliases will be available every time you start PowerShell, without having to importing aliases or manually run scripts.

We won't go into detail on profiles here, to learn more see the Owner's Manual topic on Windows PowerShell Profiles. We'll just show you quickly how simple it is to add the aliases to your profile.

Note. The example we'll be showing you works with the profile with a scope of current user, current host. There are other profiles you can work with, which are explained in the "Windows PowerShell Profiles" section of this Owner's Manual.

Start by opening your profile in Notepad, like this:

```
notepad $profile
```

Now just enter your Set-Alias commands (and any necessary functions) into your profile:

```
Set-Alias d Get-Date

function FindDefaultPrinter
{
    Get-WMIObject –query "Select * From Win32_Printer Where Default = TRUE"
}

Set-Alias dp FindDefaultPrinter
Set-Alias excel "C:\Program Files\Microsoft Office\Office12\Excel.exe"
```

Save the profile, close it, close Windows PowerShell, then open PowerShell again. Try out any of the aliases we just saved – they'll all work. Every time, no extra steps involved.

How easy was *that*?!

Alias Restrictions

Here a couple things you *can't* do with aliases. One is that you can't alias a cmdlet with specific parameters. For example, say you use this command a lot:

```
Set-Location C:\Scripts\old
```

If so, it might be useful to you to create an alias to this command. Let's give it a try:

```
Set-Alias toOld (Set-Location C:\Scripts\old)
```

Will this work? Nope. Here's what you get back:

```
Set-Alias : Cannot bind argument to parameter ‘Value’ because it is null.
At line:1 char:10
+ Set-Alias <<<<  toOld (Set-Location c:\scripts\old)
    + CategoryInfo          : InvalidData: (:) [Set-Alias],
ParameterBindingValidationException
    + FullyQualifiedErrorId : ParameterArgumentValidationErrorNullNotAllowed,Microsoft.
PowerShell.Commands.SetAliasCommand
```

If you want to do something like this, you need to put the cmdlet in a function, then alias the function:

```
function toOld{Set-Location C:\Scripts\Old}
```

Now you can either call the function toOld, or you can alias the function:

```
set-alias to toOld
```

Now typing either *toOld* or *to* will take you to the C:\Scripts\Old folder.

You also can't alias a pipelined command. Try this:

```
Set-Alias as (Get-Alias | Sort-Object)
```

It didn't work, did it? Again, you can put the command in a function and then set an alias, but you can't directly alias a pipelined command.

The End

Oh, and in case you haven't figured it out yet, the names we mentioned at the beginning of this article belong to Mark Twain, Cary Grant, Martin Sheen, and O. Henry. Makes us wonder if we should give ourselves an alias….

Windows PowerShell Profiles

It's like an Autoexec.bat file for Windows PowerShell. Although your Windows PowerShell profile can do things Autoexec.bat could only dream of doing.

You must admit, Windows PowerShell has a great profile. You hadn't noticed? Start Windows PowerShell and stand to the side of your monitor. See it now? True, you can't see much of anything on the monitor anymore, but you must be able to see that Windows PowerShell looks great from the side. Title bar not too big, coloring about right, smooth face…yep, Windows PowerShell has a great profile. You don't see any of the slight imperfections you might see while looking at it head-on.

So why would we want to mess with something like that? Well, we wouldn't. Instead we're going to explain how to set up and change a different type of profile, a profile that gives you a tremendous amount of control over your Windows PowerShell experience. This profile will help you smooth out some of those head-on imperfections – sorry, *beauty marks* – you might have noticed.

Before we get started, stop staring at the side of your monitor and sit back down in front of it. Sorry, but it really will be easier to follow along this way.

The Profile

So what is this profile we're talking about? In the simplest sense, it's a text file. A little less simple, it's a Windows PowerShell script file (a file with a .ps1 file extension). So what makes this a profile rather than just a script file? Location, location, location. And name. And – well, we'll get to all that in a moment.

The Windows PowerShell profile is simply a script file that runs when Windows PowerShell starts up. You can put cmdlets, scripts, functions – any valid Windows PowerShell commands – into this script file. Each time you start Windows PowerShell, this script file will run. That means you can use the profile to set up your Windows PowerShell environment. Typically that would be custom console settings and aliases, but use your imagination and you can come up with other things you'd like to customize in PowerShell before you start working with it.

Finding the Profile

We mentioned location (three times, which should give you some idea how important that is). What makes the profile a profile and not a regular script file is the name and location of the file. Type this at your PowerShell command prompt:

```
$profile
```

On Windows Vista and Windows Server 2008 that built-in variable will return something like this:

```
C:\Users\kenmyer\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
```

This is the full path to the file Windows PowerShell will try to run when it starts. Notice we said "try" to run. Here's an interesting fact: just because you were able to find the profile doesn't mean it actually exists. $profile is simply a built-in variable that contains the full path to where the profile will be if there is one; the file doesn't actually have to exist, and by default it doesn't. If you want a profile to run when you start Windows PowerShell, you need to create this file.

Profile Scope

Before we create our profile, we should point out that Windows PowerShell (as of version 2.0) actually recognizes multiple profiles. The profile we already showed you (you know, the one with a path stored in the $profile variable – did you forget already?) applies to the current user and current host. Notice in the example path we gave that the path starts out with Users\kenmyer. When you set this profile, it runs only when keymyer is the logged-on user. That means each user of a computer can have a unique profile.

There are also machine-specific profiles: these profiles will run for all users on a given machine. In addition, profiles can be created for specific hosts, meaning third-party PowerShell hosts can have their own profiles. If multiple profiles exist, user profiles take precedence.

Here's a list of profiles and their locations:

Scope	Variable	Path
Current User, Current Host	$profile	$home\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Current User, All Hosts	$profile.CurrentUserAllHosts	$home\Documents\WindowsPowerShell\profile.ps1
All Users, Current Host	$profile.AllUsersCurrentHost	$pshome\Microsoft.PowerShell_profile.ps1
All Users, All Hosts	$profile.AllUserAllHosts	$pshome\profile.ps1

For the rest of this discussion we're going to keep things simple and work only with the current user, current host profile ($profile). But everything we talk about applies to all of these profiles.

Creating a Profile

Before we decide to create a profile, let's check to see whether we already have one:

```
Test-Path $profile
```

If the profile exists this command will return True; if it doesn't exist, the command will return False. If this command returns False, you need to create the profile.

Creating a profile in Windows XP is really easy. Simply type this at the command prompt:

```
notepad $profile
```

This command opens the profile in Notepad. If the profile doesn't exist, you'll be prompted to create it. If you choose Yes, the file will be created for you and will be opened in Notepad. Remember, though, we said this is for Windows XP. If you're running Windows Vista or Windows Server 2008 things get a little more complicated. (But just a little bit, not very much.) If the full path to this file doesn't exist, trying to open it in Notepad will return an error; you won't receive any prompts and the file won't be created for you. The way to create this file in Windows Vista and Windows Server 2008 (and which will also work in Windows XP) is to use the New-Item cmdlet:

```
New-Item -path $profile -type file -force
```

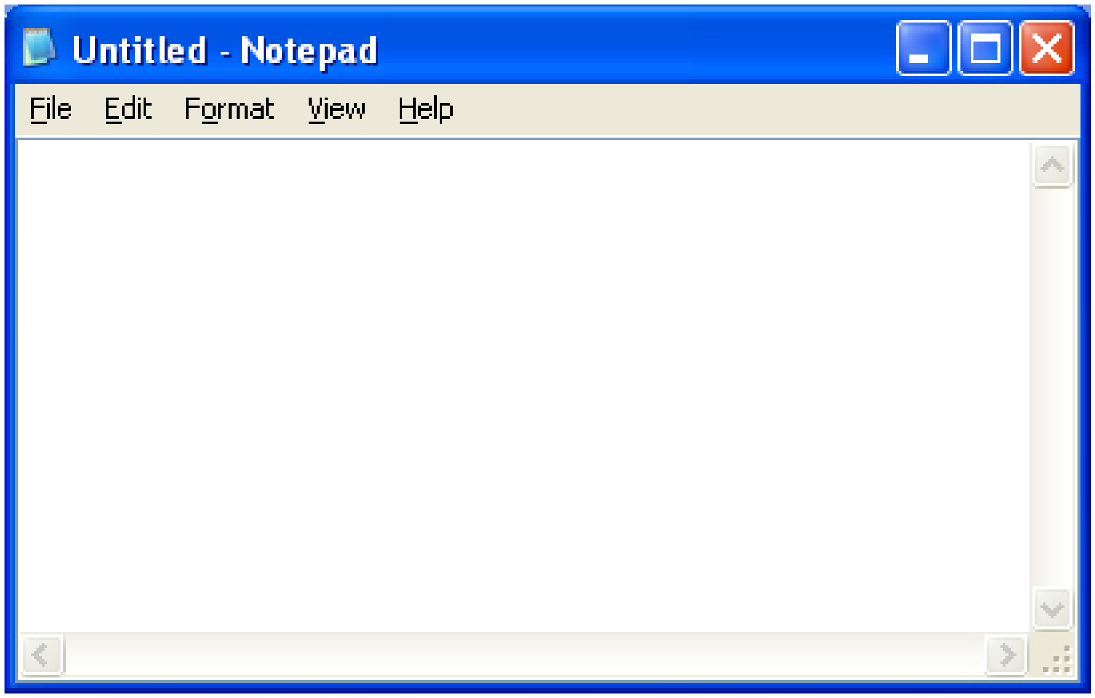
We pass three parameters to New-Item:

- **-path $profile.** We're passing the full path, stored in the $profile variable, of the item we want to create.
- **-type file.** This tells New-Item what type of item we're creating, in this case a file.
- **-force.** This parameter tells New-Item to create the full path and file no matter what.

Now you can open the profile and take a look:

```
notepad $profile
```

And what will you see? Notepad:



What Goes in the Profile?

There are a lot of different things you can put in your profile. We're going to do just a couple of simple things to show you how it works. You might notice when you start Windows PowerShell that you always start out in the same folder. For example, on Windows Vista, by default, you start in your user folder:

```
PS C:\Users\kenmyer>
```

Well, as it turns out this isn't the folder you usually work in; maybe you usually work in your C:\Scripts folder. So every single time you start Windows PowerShell the first thing you have to do is change directories to the C:\Scripts folder. Not a big deal, but kind of a hassle. Well, why not just set up your profile to start you out in the C:\Scripts folder in the first place? Give this a try:

If your profile isn't open, open it now. (Remember, we just showed you how to do that.) Inside your profile type this:

```
Set-Location C:\Scripts
```

Save the profile and close Windows PowerShell. Now open PowerShell again. Voila!

Now every time you open PowerShell, you'll go straight to your C:\Scripts folder. You can also do things such as set aliases and run functions. Not only that, but it's simple to set up all these things on other computers; simply copy the commands from your profile to the profile on the other machine.

For an example of other things you can do with profiles, take a look at **Customizing the Windows PowerShell Console** in this Owner's Manual.

Wait, My Profile Didn't Run

What's that? You say you opened a new instance of Windows PowerShell and all you got was an error message? You mean an error message like this?

```
File C:\Users\kenmyer\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
cannot be loaded because the execution of scripts is disabled on this system. Please
see "get-help about_signing" for more details.
```

There's a very good explanation for this: you haven't set your execution policy. We talk a little more about this in the **Introduction to Scripting** section of this Owner's Manual, but here's a quick explanation.

By default, Windows PowerShell does not allow scripts to run within the console window. And, since the profile is nothing more than a script file that runs automatically when PowerShell starts up, the profile can't run. In order to allow your profile to run you need to set your execution policy to a level that allows scripts to run. First, check your execution policy by typing this cmdlet at the command prompt:

```
Get-ExecutionPolicy
```

By default this will return a value of Restricted. To allow scripts to run, use the Set-ExecutionPolicy cmdlet to change the policy to something like this:

```
Set-ExecutionPolicy Bypass
```

Close your Windows PowerShell window and reopen it. Now your profile should run just fine.

Now if you'd like to go back to looking at Windows PowerShell in profile, go right ahead. But you'll probably find it much more useful to remain head-on and work with the Windows PowerShell profile files.

Scripting

Need to perform the same Windows Powershell task over and over and over again? Ever thought about putting the commands for carrying out this task into a script?

In this world there's always something to be glad about. (Hey, it's true – just ask Pollyanna.) For example, system administration can sometimes be a complicated job that requires a lot of time and effort. But we can all be glad that we have Windows PowerShell to make the job easier. Sometimes PowerShell requires that you type a lot of information to the command line. But we can be glad we have shortcuts (see **Windows PowerShell Shortcuts**) and aliases (see **Windows PowerShell Aliases**) to make this quicker and simpler. And we can be especially glad that we have Windows PowerShell scripts.

Suppose you find yourself typing the same commands over and over practically every time you start up Windows PowerShell. Wouldn't it be much easier to simply type a file name to the command prompt than to retype this possibly long and complicated command over and over? Of course it would. Or suppose you want to schedule a command to run at a particular time. You can't always be there are 2:00 AM to type in the command, but with a script you can schedule the command. Or, as another example, maybe you have a long series of commands that need to be run in sequence and perform a somewhat complicated set of actions. A script will make this a very simple task to accomplish.

In this section we're going to explain everything you need to know to get started writing scripts in Windows PowerShell. But before we start writing scripts, you need to know how some of the pieces work.

Objects, Properties, and Variables

If you read the **Getting Started with Windows PowerShell** section of this Owner's Manual you know (or at least you should know) that cmdlets are the heart and soul of Windows PowerShell. What we didn't tell you – because you were just getting started we didn't want to scare you away – was that there's a basic principal of software development that lies behind the workings of every cmdlet: the object.

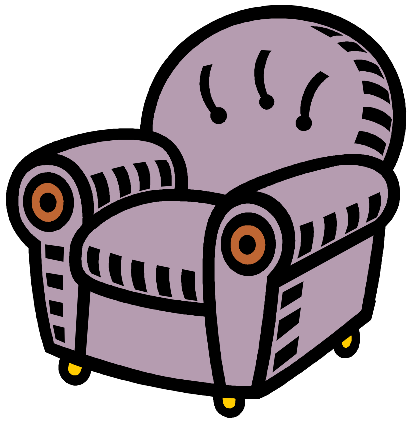
Hang on, don't go yet. Yes, we said "software development," and you're not a software developer, you're a system administrator. Big difference, whole different skill set, we know. But don't worry, you don't need to be a developer to work with PowerShell (that would kind of defeat the whole purpose, wouldn't it?), and you don't need to be a developer to understand objects. But as a system administrator who's going to be working with Windows PowerShell, objects are a very useful thing to know about.

So your first question is probably ”Why do I need to know about objects?” (Either that or “What is an object?”, but we’re sure you’ll quickly get around to the “Why do I care?” question.) If we do our jobs right (which is never a guarantee, but in this case we’ll try), the reason you need to know will become clear as you read through this. As to what an object is, well, that’s easy: an object is a thing.

See, that was painless, wasn’t it?

What? What kind of thing? Well…anything. A boat is an object. A chair is an object. A donut is an object. (Mmmm, donuts.) A process is an object. A shoe is an object. A –

Huh? Yes, as a matter of fact we *did* say that a process is an object. Okay, let’s start over. We’ll start with the chair. (We’d start with the donut, but some of us tend to lose our focus when donuts are mentioned.) A chair is an object – it’s a thing that exists and has various qualities. Take a look at our chair:



As we mentioned, an object has various qualities. Let’s look at some of the qualities of this chair: it has a color; it has arms; it has feet; it has a cushion; it has a back. These are all qualities that you can use to identify this particular chair, like this:

Color	Purple
Arms	True
Legs	4
Cushion	True
Back	True

In the world of computers, these qualities are known as *properties*. For this particular chair, the Color property has a value of Purple. You might prefer to choose a chair with a color property of Green. A chair either has arms or it doesn’t: this chair does, so the Arms property is True. Some stools have only three legs, while some rolling chairs don’t actually have legs, they have wheels. Our chair has four legs, so our Legs property has a value of 4.

Computers work the same way. A process is a thing: it has qualities, or properties, that identify that particular process. Try running the Get-Process cmdlet in Windows PowerShell. Run this from the command prompt and press Enter:

```
Get-Process
```

Depending on the processes running on your computer, you'll get output similar to the following:

```
Handles  NPM(K)     PM(K)       WS(K) VM(M)   CPU(s)      Id ProcessName
-------  ------     -----       ----- -----   ------      -- -----------
     71       3      1640        5544    52     0.08    2276 AcroBroker
     27       1       376        1616    11             1136 AEADISRV
     38       2       732        2348    21             1332 agrsmsvc
     57       3      1480        4944    53     0.06    3472 apdproxy
    105       3      1100        4108    37             1172 Ati2evxx
    137       4      1820        6216    42             1636 Ati2evxx
    111       4     11812       14520    43             1352 audiodg
    314      12     22684        3536   158     0.48    2824 CCC
    517      19     33476       11236   198     2.37    4400 CCC
   1066      16     38848       50692   148             2956 CcmExec
   1219      27     34580       49268   250    24.94    3936 communicator
    723       6      1752        5528    86              560 csrss
…
```

The output shows you some of the properties of a process. For example, you can see that each process has a Name. The first process in the list has a Name of AcroBroker. A process also has an Id; here the first process in the list has an Id of 2276. And so on.

We've seen in other parts of this Owner's Manual that you can find objects based on specific properties by using cmdlet parameters. For example, if you want to find the process with the Id 2824 you'd type this at the command prompt:

```
Get-Process –Id 2340
```

And, at least on our test machine, this is what we got back:

```
Handles  NPM(K)     PM(K)       WS(K) VM(M)   CPU(s)      Id ProcessName
-------  ------     -----       ----- -----   ------      -- -----------
    125       5      5928        9744    60             2340 svchost
```

Another way to work with properties is by assigning the output of your commands to a variable. A variable is simply a place to save data in memory so you can later manipulate that data. In Windows PowerShell, all variables begin with a dollar sign ($). Let's assign the results of our Get-Process call to a variable we'll call $p:

```
$p = Get-Process –Id 2340
```

The first thing you'll notice when you run this command is that you don't get any output. (Keep in mind you might not have a process with Id 2340 on your computer. Run Get-Process and choose an Id that will work on your computer.) That's because, instead of going to the display, all the information about this process object has been stored in the variable $p. How do we know this for sure? Just type this at the command prompt:

```
$p
```

Surprise! There's the output. Now let's find out what the name of this process is. To do that, we simply add a dot (.) plus the name of the property to the end of our variable name, like this:

```
$p.Name
```

Type this at the command prompt and you'll see the name of the process with the Id 2340.

You can also use variables for values other than objects. For example, type this at the command prompt:

```
$a = 2 + 3
```

Now type $a at the command prompt and press Enter. Here's what you'll see:

```
5
```

You can continue to use this variable for as long as this Windows PowerShell session is open. Try this:

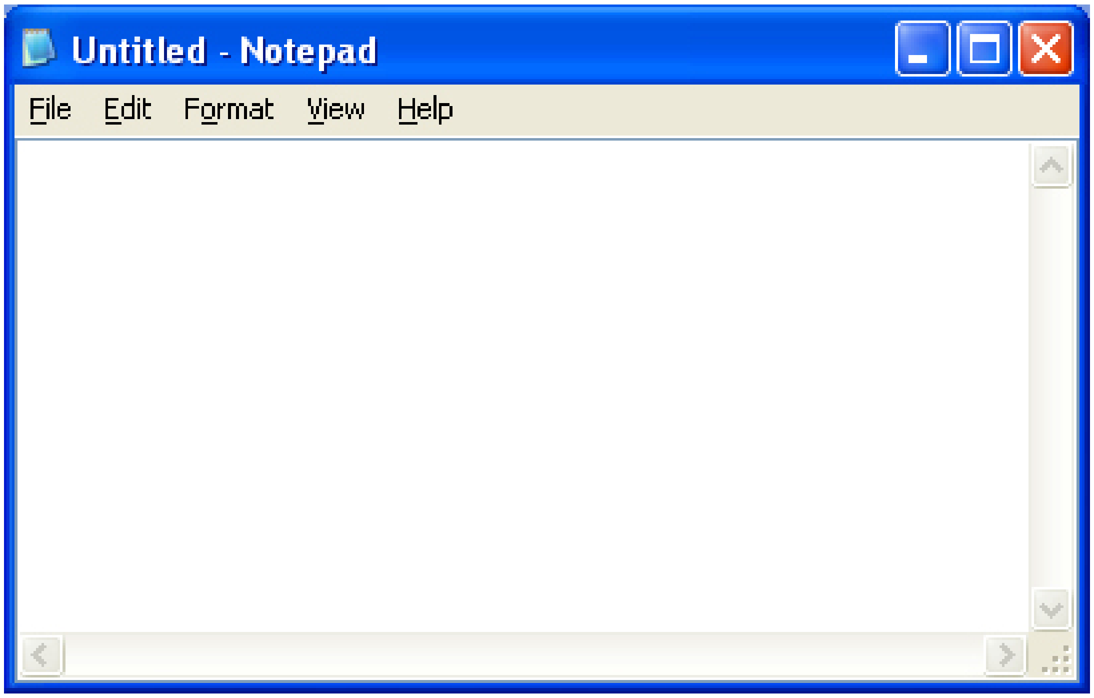
```
PS C:\scripts> $b = 8
PS C:\scripts> $b - $a
3
```

Aren't you glad we showed you that?

Creating a Script

Now that we have a general idea of how objects, properties and variables work, let's talk a little bit about actually creating a script. A script in Windows PowerShell is simply a text file with a .ps1 file extension. Let's create our first script.

Start by opening Notepad. (Simply type notepad.exe at the PowerShell command prompt; or, from the Start menu, select All Programs, Accessories, then select Notepad.) You'll have an empty Notepad window:



All you need to do here is type in the commands you would normally type at the command prompt. For example, type this into Notepad:

```
$a = Get-Process –Name svchost
Write-Host "Here are all the processes with the name svchost:"
$a
```

Before you run this script you need to save it. Select Save from the File menu (or press Ctrl+S). In the Save dialog box, enter the name you'd like to save the script to. Before you select Save, however (you didn't just go ahead and try to save did you?), make sure you save the file with a .ps1 file extension. If you simply type test.ps1 in the File Name box of the Save dialog box you'll end up with a file with the name test.ps1.txt. That's because Notepad will helpfully assume that you're saving a text file and will append the .txt extension for you. To keep this from happening, either enclose the file name in double quotes ("test.ps1") or type test.ps1 in the File Name box and select All Files (*.*) from the Save As Type drop-down list box.

Note: Keep in mind that what we showed you here isn't the only thing you can do in a script. Anything you can do from the command prompt you can put in a script. A script can be anywhere from one line to hundreds of lines long. It all depends on what you want it to do and how complicated you want to get.

You've just created your very first Windows PowerShell script. Congratulations! Aren't you glad you know how to write a script now?

Running a Script

Now that you've created your script, you probably want to run it and see it in action. So you open up Windows Explorer to the folder where you saved your script file, you double-click the .ps1 file, sit back, and wait for the magic to happen.

As it turns out, however, *this* is what happens:

Hmmm, instead of running, your script opened up in Notepad. Interesting, but not exactly what you had in mind. Oh wait, you think, I get it: you probably have to run Windows PowerShell before you can run a Windows PowerShell script. OK, that makes sense. And so, with that in mind, you open up Windows PowerShell and type the path to the .ps1 file at the command prompt. You press ENTER and wait for the magic to happen.

As it turned out, however, *this* is what happens:

```
File C:\scripts\test.ps1 cannot be loaded because the execution of scripts is disabled
on this system. Please see "get-help about_signing" for more details.
```

Wow; how nice. A new command shell and scripting environment that doesn't even let you run scripts. What will those guys at Microsoft think of next?

Listen, don't panic; believe it or not, everything is fine. You just need to learn a few little tricks for running Windows PowerShell scripts.

Running Scripts From Within Windows PowerShell

Let's start with running scripts from within Windows PowerShell itself. (Which, truth be told, is probably the most common way to run Windows PowerShell scripts.) Why do you get weird error messages when you try to run a script? That's easy. The security settings built into Windows PowerShell include something called the "execution policy;" the execution policy determines how (or if) PowerShell runs scripts. By default, PowerShell's execution policy is set to **Restricted**; that means that scripts – including those you write yourself – won't run. Period.

Note. You can verify the settings for your execution policy by typing the following at the PowerShell command prompt and then pressing ENTER:

```
Get-ExecutionPolicy
```

Now, admittedly, this might seem a bit severe. After all, what's the point of having a scripting environment if you can't even run scripts with it? But that's OK. If you don't like the default execution policy (and you probably won't) then just go ahead and change it. For example, suppose you want to configure PowerShell to run – without question – any scripts that you write yourself, but to run scripts downloaded from the Internet only if those scripts have been signed by a trusted publisher. In that case, use this command to set your execution policy to **RemoteSigned**:

```
Set-ExecutionPolicy RemoteSigned
```

Alternatively, you can set the execution policy to **AllSigned** (all scripts, including those you write yourself, must be signed by a trusted publisher) or **Bypass** (*all* scripts will run, regardless of where they come from and whether or not they've been signed).

See? No need to panic at all, is there?

Note. Not sure what we mean by "signing scripts?" Then open up PowerShell, type the following, and press ENTER:

```
Get-Help About_Signing
```

After you change your execution policy settings it's *possible* to run scripts. However, you still might run into problems. For example, suppose you change directories from your Windows PowerShell home directory to C:\Scripts (something you can do by typing **cd C:\Scripts**). As it turns out, the C:\Scripts folder contains a script named Test.ps1. With that in mind you type the following and then press ENTER:

```
Test.ps1
```

And here's the response you get:

```
The term 'test.ps1' is not recognized as a cmdlet, function, operable program, or
script file. Verify the term and try again.
```

We know what you're thinking: didn't we just change the execution policy? Yes, we did. However, this has nothing to do with the execution policy. Instead, it has to do with the way that PowerShell handles file paths. In general, you need to type *the complete file path* in order to run a script. That's true regardless of your location within the file system. It doesn't matter if you're in C:\Scripts; you still need to type the following:

```
C:\Scripts\Test.ps1
```

Now, we said "in general" because there are a couple of exceptions to this rule. For example, if the script happens to live in the current directory you can start it up using the **.** notation, like so:

```
.\Test.ps1
```

Note. There's no space between the .\ and the script name.

And while PowerShell won't search the current directory for scripts it *will* search all of the folders found in your Windows PATH environment variable. What does that mean? That means that, if the folder C:\Scripts is in your path, then you *can* run the script using this command:

```
Test.ps1
```

But be careful here. Suppose C:\Scripts is *not* in your Windows path. However, suppose the folder D:\Archive *is* in the path, and that folder also contains a script named Test.ps1. If you're in the C:\Scripts directory and you simply type **Test.ps1** and press ENTER, guess which script will run? You got it: PowerShell won't run the script in C:\Scripts, but it *will* run the script found in D:\Archive. That's because D:\Archive is in your path.

Just something to keep in mind.

Note. Just for the heck of it, here's a command that retrieves your Windows PATH environment variable and displays it in a readable fashion:

```
$a = $env:path; $a.Split(";")
```

Even More About File Paths

Now we know that all we have to do is type in the full path to the script file and we'll never have to worry about getting our scripts to run, right? Right.

Well, *almost* right. There's still the matter of scripts whose path name includes a blank space. For example, suppose you have a script stored in the folder C:\My Scripts. Try typing this command and see what happens:

```
C:\My Scripts\Test.ps1
```

Of course, by now you've come to expect the unexpected, haven't you? Here's what you get back:

```
The term 'C:\My' is not recognized as a cmdlet, function, operable program, or script
file. Verify the term and try again.
```

This one you were able to figure out on your own, weren't you? Yes, just like good old Cmd.exe, PowerShell has problems parsing file paths that include blank spaces. (In part because blank spaces are how you separate command-line arguments used when you started the script.) In Cmd.exe you can work around this problem by enclosing the path in double quotes. Logically enough, you try the same thing in PowerShell:

```
"C:\My Scripts\Test.ps1"
```

And here's what you get back:

```
C:\My Scripts\Test.ps1
```

Um, OK …. You try it again. And here's what you get back:

```
C:\My Scripts\Test.ps1
```

You try it – well, look, there's no point in trying it again: no matter how many times you try this command, PowerShell will simply display the exact same string value you typed in. If you actually want to *execute* that string value (that is, if you want to run the script whose path is enclosed in double quotes) you need to preface the path with the Call operator (the ampersand). You know, like this:

```
& "C:\My Scripts\Test.ps1"
```

Note. With this particular command you can either leave a space between the ampersand and the path name or not leave a space between the ampersand and the path name; it doesn't matter.

To summarize, *here's* how you run scripts from within Windows PowerShell:

- Make sure you've changed your execution policy. By default, PowerShell won't run scripts at all, no matter how you specify the path.

- To run a script, specify the entire file path, or either: 1) use the .\ notation to run a script in the current directory or 2) put the folder where the script resides in your Windows path.
- If your file path includes blank spaces, enclose the path in double quote marks and preface the path with an ampersand.

And, yes, that all takes some getting used to. However, you *will* get used to it. (To make life easier for you, we recommend that you keep all your scripts in one folder, such as C:\Scripts, and add that folder to your Windows path.)

Note. So can you use *PowerShell* to add a folder to your Windows Path? Sure; here's a command (that we won't bother to explain in this introductory article) that tacks the folder C:\Scripts onto the end of your Windows path:

```
$env:path = $env:path + ";c:\scripts"
```

Bonus: "Dot Sourcing" a Script

Admittedly, up to this point the news hasn't been all that good: you can't run a PowerShell script by double-clicking the script icon; PowerShell doesn't automatically look for scripts in the current working directory; spaces in path names can cause all sorts of problems; etc. etc. Because of that, let's take a moment to talk about one very cool feature of Windows PowerShell scripting: dot sourcing.

Suppose we have a very simple PowerShell script like this one:

```
$a = 5
$b = 10
$c = $a + $b
```

Suppose we run this script, then type **$c** at the command prompt. What do you think we'll get back? If you guessed nothing, then you guessed correctly:

In other words, we don't get back anything at all. Which, again, should come as no great surprise. And we know what you're thinking, you're thinking: "Come on, shouldn't this be leading us somewhere?"

Yes, it should. And believe it or not, it is. Let's run our PowerShell script again, only this time let's "dot source" it; that is, let's type a period and a blank space and *then* type the path to the script file. For example:

```
. c:\scripts\test.ps1
```

When we run the script nothing will *seem* to happen; that's because we didn't include any code for displaying the value of $C. But now try typing $C at the command prompt . Here's what you'll get back:

```
15
```

Good heavens! Was this a lucky guess on the part of the PowerShell console, or is this some sort of magic?

Surprisingly enough, it's neither. Instead, *this* is dot sourcing. When you dot source a script (that is, when you start the script by prefacing the path to the script file with a dot and a blank space) any variables used in the script become global variables that are available in multiple scopes. What does *that* mean? Well, a script happens to represent one scope; the console window happens to represent another scope. We started the script Test.ps1 by dot sourcing it; that means that the variable $C remains "alive" after the script ends. In turn, that means that this variable can be accessed via the command window. In addition, these variables can be accessed from other scripts. (Or at least from other scripts started from this same instance of Windows PowerShell.)

Suppose we have a second script (Test2.ps1) that does nothing more than display the value of the variable $c:

```
$c
```

Look what happens when we run Test2.ps1 (even if we don't use dot sourcing when starting the script):

```
15
```

Cool. Because $c is a global variable everyone has access to it.

And, trust us here: this *is* pretty cool. For example, suppose you have a database that you periodically like to muck around with. If you wanted to, you could write an elaborate script that includes each and every analysis you might ever want to run on that data. Alternatively, you could write a very simple little script that merely connects to the database and returns the data (stored in a variable). If you dot source that script on startup you can then sit at the command prompt and muck around with the data all you want. That's because you have full access to the script variables and their values.

Note. OK, sure, this *could* cause you a few problems as well, especially if you tend to use the same variable names in all your scripts. But that's OK; if you ever need to wipe out the variable $C just run the following command (note that, with the **Remove-Variable** cmdlet, we need to leave off the $ when indicating the variable to be removed):

```
Remove-Variable C
```

Play around with this a little bit and you'll start to see how useful dot sourcing can be.

Running Scripts Without Starting Windows PowerShell

We realize that it's been awhile, but way back at the start of this article we tried running a Windows PowerShell script by double-clicking a .PS1 file. That didn't go quite the way we had hoped: instead of running the script all we managed to do was open the script file in Notepad. Interestingly enough, that's the way it's *supposed* to work: as a security measure you can't start a PowerShell script by double-clicking a .PS1 file. So apparently that means that you *do* have to start PowerShell before you can run a PowerShell script.

In a somewhat roundabout way, that's technically true. However, that doesn't mean that you can't start a PowerShell script from a shortcut or from the **Run** dialog box; likewise you *can* run a PowerShell script as a scheduled task. The secret? Instead of calling the script you need to call the PowerShell executable file, and then pass the script path as an argument to PowerShell.exe. For example, in the **Run** dialog box you might type a command like this**:**

```
powershell.exe -noexit &'c:\scripts\test.ps1'
```

There are actually three parts to this command:

- **Powershell.exe**, the Windows PowerShell executable.
- **-noexit,** an optional parameter that tells the PowerShell console to remain open after the script finishes. Like we said, this is optional: if we leave it out the script will still run. However, the console window will close the moment the script finishes, meaning we won't have the chance to view any data that gets displayed to the screen. Incidentally, the –noexit parameter must immediately follow the call to the PowerShell executable. Otherwise the parameter will be ignored and the window will close anyway.
- **C:\Scripts\Test.ps1**, the path to the script file.

What if the path to the script file contains blank spaces? In that case you need to do the ampersand trick we showed you earlier; in addition, you need to enclose the script path in *single* quote marks, like so:

```
powershell.exe -noexit &'c:\my scripts\test.ps1'
```

Strange, but true!

Note. Here's an interesting variation on this same theme: instead of starting PowerShell and asking it to run a particular script you can start PowerShell and ask it to run a particular *command*. For example, typing the following in the **Run** dialog box not only starts PowerShell but also causes it to run the Get-ChildItem cmdlet against the folder C:\Scripts:

```
powershell.exe -noexit get-childitem c:\scripts
```

It's possible to get even *more* elaborate when starting Windows PowerShell, but this will do for now. If you'd like more information on PowerShell startup options just type **powershell.exe /?** from either the Windows PowerShell or the Cmd.exe command prompt.

So is there a catch here? Unfortunately, there is. If you are trying to carry out a task that requires administrator privileges then you can't start Windows PowerShell from the **Run** dialog box on either Vista or Windows Server 2008. OK, check that: you can *start* PowerShell, but the command you are trying to run will fail. For example, if this is your script, it will fail; that's because changing the execution policy requires administrator privileges:

```
Set-ExecutionPolicy Unrestricted
```

Fortunately, there's a workaround. If you'd like to *always* be able to start PowerShell from the **Run** dialog box on Vista or Windows Server 2008 then you should check out the Script Elevation PowerToys for Windows Vista (http://technet.microsoft.com/en-us/magazine/2008.06.elevation.aspx).

See? That Wasn't So Bad

Admittedly, running Windows PowerShell scripts might not be as straightforward and clear-cut as it could be. On the other hand, it won't take you long to catch on, and you'll soon be running PowerShell scripts with the best of them. Most important, you'll also be able to say things like, "You know, you really ought to dot source that script when you run it." If *that* doesn't impress your colleagues then nothing will.

Aren't you glad you know how to do all this now?

*"*If you'd like to *always* be able to start PowerShell from the **Run** dialog box on Vista or Windows Server 2008 then you should check out the Script Elevation PowerToys for Windows Vista *"*

Remoting

Just in case you have a computer or two that isn't located in your office.

According to the old joke, the three most important things in real estate are location, location, location. And in Windows PowerShell 2.0 the three most important things are –

Well, to tell, you the truth, we don't actually *know* what the three most important things are in Windows PowerShell 2.0. What we *do* know is that the three most important things in Windows PowerShell 2.0 are *not* location, location, location; that's because location doesn't matter in Windows PowerShell 2.0. Not in the least. (Well, except with profiles But we're not talking about profiles right now.)

Admittedly, that wasn't the case with the original release of Windows PowerShell; in PowerShell 1.0 location – or, to be more specific – *your* location – was extremely important. That's because Windows PowerShell 1.0 did not support (except in one or two minor instances) the concept of remoting, the ability for you to sit at computer A and retrieve information from, or make changes to, computer B. The only way for you to retrieve information from computer B was to physically sit at computer B. The idea of sitting at your desk and managing all your computers simply by transmitting commands across the network? As they say in New York, fuhhgeddaboutit.

Note. If you're not from New York, that can be translated like this: forget about it.

Now, that might not have been all that bad except for one thing: Windows PowerShell was being promoted as *the* tool for system administration. System administration that could only be done locally, and only on one computer at a time. As they say in New York –

Well, you're right: they say a lot of things in New York, don't they? Never mind.

But that was then and this is, um, not then. With Windows PowerShell 2.0 you can now sit at computer A and manage computer B; for that matter, you can also manage computers C, D, E, F, and G. Remoting is now a real and viable part of Windows PowerShell; in fact, PowerShell now features *three* different ways to manage remote computers:

- **By using .NET remoting**. A handful of Windows PowerShell cmdlets (including **Get-Service** and **Get-Event**) harness the remoting capabilities built into the .NET Framework; that means that you can use these cmdlets to directly return data from a remote computer.

(As opposed to making a transient or persistent connection.)

- **By making a transient connection to a remote computer**. With a transient connection you make a temporary connection to a remote computer. How temporary is this connection? *Very* temporary: you make the connection, run a single command, and then the connection is automatically terminated.
- **By making a persistent connection to a remote computer**. With a persistent connection you make a connection to a remote computer and then that connection remains open until you explicitly close it. That means you can run as many commands as you like, without ever having to open a new connection.

In this section of the *Owner's Manual*, we'll give you a quick introduction to all three methods of managing remote computers.

What do I need in order to make these remote connections?

To tell you the truth, not much. Obviously you need Windows PowerShell 2.0. Although PowerShell 2.0 hasn't been officially released yet, you can download an RC (Release Candidate) version from https://connect.microsoft.com/windowsmanagement/Downloads. This isn't the *official* release of PowerShell 2.0, but it's very close. This RC, called the Windows Management Framework, includes Windows PowerShell 2.0, Windows Remote Management (WinRM) 2.0, and Background Intelligent Transfer Service (BITS) 4.0.

In addition, you'll need version 2.0 of the .NET Framework; there's a pretty good chance you already have this installed in your computer.

And keep in mind that this software not only needs to be installed on your computer, but it also needs to be installed on every computer that you want to be able to manage remotely. Do you want to be able to manage all your client computers using PowerShell 2.0? Then PowerShell 2.0, .NET 2.0, and WinRM must be installed on each of those client computers.

Oh, and did we mention that you need to be a local administrator on your computer and on each remote computer you try to connect to? Well, we should have, because that's absolutely crucial.

How does remoting work?

In a nutshell, *here's* how remoting works in Windows PowerShell 2.0. To begin with, you make a connection between your computer and a remote computer. You then type a command on your computer and that command is transmitted across the network to the remote computer. (In case you're worried, don't be: all these transmissions are encrypted and secure.) The command is then executed on the remote computer. Note, however, that the command runs "invisibly;" nothing happens on the remote computer to indicate that the computer is running a Windows PowerShell command. When the command completes, the output is converted to XML format and transmitted back to your computer. Your computer then converts that XML packet back into a Windows PowerShell object.

The fact that PowerShell remoting transmits XML (or, to be more specific, SOAP packets) explains why this type of remoting is considered firewall-friendly, and can be carried out across the Internet. Previous system administration tools relied on DCOM (distributed COM) in order to do remoting; with DCOM, objects are transmitted across the network. That's a problem: by default, most firewalls are designed to block objects. However, most firewalls *are* configured to allow XML packets; thus PowerShell remoting can typically be used without additional firewall configuration, and without allowing potentially-unsafe objects onto your network.

Can we get started now?

OK, good point, maybe it *is* time to actually start doing something. Let's start by taking a minute to discuss .NET remoting, a type of remoting that – as we noted – applies to only a few cmdlets.

Note. Which cmdlets? In general, cmdlets that support the **–ComputerName** parameter support .NET remoting. You can retrieve a list of cmdlets that support the –ComputerName parameter by typing the following command at the Windows PowerShell prompt and then pressing ENTER:

```
Get-Help * -Parameter ComputerName
```

By default, when you run a cmdlet like **Get-Service** you simply, well, run the cmdlet, like so:

```
Get-Service
```

Do that, and Get-Service will retrieve information about all the services running on the local computer:

```
Status   Name               DisplayName
------   ----               -----------
Stopped  Adobe LM Service   Adobe LM Service
Stopped  AdobeActiveFile... Adobe Active File Monitor V4
Stopped  Alerter            Alerter
Running  ALG                Application Layer Gateway Service
Running  Apple Mobile De... Apple Mobile Device
Stopped  AppMgmt            Application Management
Running  ASChannel          Local Communication Channel
Stopped  aspnet_state       ASP.NET State Service
Stopped  Ati HotKey Poller  Ati HotKey Poller
Running  AudioSrv           Windows Audio
…
```

That's simple enough. But what if you want to run that command against a *remote* computer? As it turns out, that's simple enough, too: you just add the **–ComputerName** parameter followed by the name (or the IP address, or the fully qualified domain name) of the remote computer. For example, this command returns information from the remote computer atl-ocs-001:

```
Get-Service –ComputerName atl-ocs-001
```

Run *this* command, and you'll get back something similar to this:

```
Status   Name               DisplayName
------   ----               -----------
Stopped  Adobe LM Service   Adobe LM Service
Stopped  AdobeActiveFile... Adobe Active File Monitor V4
Stopped  Alerter            Alerter
Running  ALG                Application Layer Gateway Service
Running  Apple Mobile De... Apple Mobile Device
Stopped  AppMgmt            Application Management
Running  ASChannel          Local Communication Channel
Stopped  aspnet_state       ASP.NET State Service
Stopped  Ati HotKey Poller  Ati HotKey Poller
Running  AudioSrv           Windows Audio
```

As you can see, the output – well, now that you mention it, the output *does* look a lot like the output we got when we ran Get-Service without the –ComputerName parameter, doesn't it? There's actually two reasons for that. For one, we *are* retrieving service information from the remote computer; needless to say, service information from one computer will always resemble service information from another computer. In addition to that, Get-Service typically suppresses the value of the **MachineName** property, a property that indicates which computer the service is running on. But that's OK; we can use a command like this to ask Get-Service to show us just the values of the DisplayName and the MachineName properties:

```
Get-Service –ComputerName atl-ocs-001 | Select-Object DisplayName, MachineName
```

Now take a peek at what we get back:

```
DisplayName                                   MachineName
-----------                                   -----------
Adobe LM Service                              atl-ocs-001
Adobe Active File Monitor V4                  atl-ocs-001
Alerter                                       atl-ocs-001
Application Layer Gateway Service             atl-ocs-001
Apple Mobile Device                           atl-ocs-001
Application Management                        atl-ocs-001
Local Communication Channel                   atl-ocs-001
ASP.NET State Service                         atl-ocs-001
Ati HotKey Poller                             atl-ocs-001
Windows Audio                                 atl-ocs-001
Background Intelligent Transfer Ser...        atl-ocs-001
…
```

Not bad, huh? Incidentally, you aren't limited to passing just one machine name to the –ComputerName parameter. Want to retrieve service information from computers atl-ocs-001, atl-ocs-002, and atl-ocs-003? Then just ask Get-Service to retrieve service information from each of these three machines, separating the computer names with commas:

```
Get-Service –ComputerName atl-ocs-001,atl-ocs-002,atl-ocs-003
```

Give that a try and see what happens.

Making a transient connection

As we noted, .NET remoting is supported on only a handful of cmdlets. That's great, but what if you need to use a cmdlet that *doesn't* support .NET remoting? Are you just plain out of luck?

Let's hope not; otherwise this is going to a very disappointing chapter. (Of course, it might be anyway. But that has nothing to do with .NET remoting.) Fortunately for all of us, the answer to that question is: no, you are *not* out of luck. As long as you can make a connection to a remote computer you can run pretty much any cmdlet you want against that remote computer.

Which leads to a very obvious follow-up question: so how *do* you make a connection to a remote computer? Remember awhile back when we described transient connections, a temporary connection that enables you to run a single command against a remote computer? Well, here's how you make a transient connection to a remote computer:

```
Invoke-Command -ComputerName atl-ocs-001 -ScriptBlock {Get-PSDrive}
```

Let's see if we can figure out how this command works. We start by calling **Invoke-Command**, the cmdlet that enables us to make a connection to a remote computer and then run a single command against that computer. To indicate the name of that remote computer, we add the **–ComputerName** parameter followed by the name (or, again, the IP address or the fully qualified domain name) of the remote computer. As you can see, in this example that's *atl-ocs-001*.

As for which command we're going to run against atl-ocs-001, we indicate *that* by including the **–ScriptBlock** parameter followed by the command to be run; in this case, that's the **Get-PSDrive** cmdlet. Note that the command must be enclosed in curly braces; in Windows PowerShell, the curly braces tell the system that the item inside the braces is a command to be executed rather than a text value.

And, believe it or not, that's all there is to it. (OK, there are additional parameters that do things such as enable you to run the command under alternate credentials, but we're going to skip those parameters for now.) When we run this command we should get back something similar to this:

```
Name       Provider      Root        CurrentLocation PSComputerName
----       --------      ----        --------------- --------------
A                        A:\                         atl-ocs-001
Alias                                                atl-ocs-001
C                        C:\                         atl-ocs-001
cert                     \                           atl-ocs-001
Env                                                  atl-ocs-001
Function                                             atl-ocs-001
HKCU                     HKEY_CU...                  atl-ocs-001
HKLM                     HKEY_LO...                  atl-ocs-001
Variable                                             atl-ocs-001
WSMan                                                atl-ocs-001
```

There are two things of particular to note in this output. First, take a peek at the **PSComputerName** property. This is a property that Invoke-Command automatically adds to returned data; as you might have guessed, it tells us where that data came from. As you can see, all the retrieved data originated on the computer atl-ocs-001.

Second, take a look at the values returned for the **CurrentLocation** property. CurrentLocation tells us, well, the current location within the PowerShell drive; for example, if you were working in the PowerShell console and your current folder was C:\Scripts then the CurrentLocation for drive C would be C:\Scripts. So then why are all the CurrentLocation fields blank in our output? That's easy: that's because this data came from a remote computer, and our command was running in an invisible window rather than in the Windows PowerShell console. PowerShell wasn't up and running, so there *weren't* any current locations.

See? We *told* you this data came from a remote computer.

Even better, you can make transient connections against multiple computers. All you have to do is add all the computer names to the –ComputerName parameter, just like we did with .NET remoting:

```
Invoke-Command -ComputerName atl-ocs-001,atl-ocs-002,atl-ocs-003 `
 -ScriptBlock {Get-PSDrive}
```

Here's another cool little trick. What if you'd like to get information from the *local* computer as well as from a remote computer (or computers)? Well, that's no problem; you can always include the name of the local computer as one of the computer names supplied to the –ComputerName parameter. But if you want to be cool (and, let's face it, who *doesn't* want to be cool?) you could do one of these two things instead;

- Use a dot (.) instead of the local computer name.
- Use *localhost* instead of the local computer name.

In other words, this command runs Get-PSDrive against the local computer:

```
Invoke-Command -ComputerName . -ScriptBlock {Get-PSDrive}
```
Oh, and so does this one:

```
Invoke-Command -ComputerName localhost -ScriptBlock {Get-PSDrive}
```

Making a persistent connection

The one problem with a transient connection is that it's transient: as soon as your one command finishes executing your connection is closed. Need to run another command? Then you need to open a second connection. Need to run a third command? Then you need to open a third – well, you get the idea. Transient connections work for simple tasks; they're less useful for more complex tasks. For something like that you need a connection that's a bit more persistent. You know, like a persistent connection. Or something.

To explain how a persistent connection works let's walk you through a simple example. In order to make a persistent connection you need to create a new PowerShell (PS) session. The easiest way to do that is to use a command similar to this (in case you're wondering why we left in the PowerShell prompt – **PS C:\Scripts>** – well, you'll find out in just a second):

```
PS C:\Scripts> Enter-PSSession -ComputerName atl-ocs-001
```

As you can see, all we've done here is call the **Enter-PSSession** cmdlet along with the **–ComputerName** parameter; the value passed to –ComputerName is – oh, you guessed it. Well, you're right: it's the name of the remote computer.

So what's going to happen when we run this command? This is going to happen:

```
[atl-ocs-001]: PS C:\Users\Administrator\Documents>
```

As you can see, our prompt has changed, and in two very important ways. First, we're now working on the remote computer atl-ocs-001; notice how the prompt now begins with **[atl-ocs-001]** to let us know which computer we're on. Second, note that we're now working in the C:\Users\Administrator\Documents folder. But remember, that's the C:\Users\Administrator\Documents folder on the remote computer atl-ocs-001. From now on (at least until we end our session) everything we type, and every command we run, will execute on atl-ocs-001.

Here's an example. The following command retrieves the value of the environment variable ComputerName, an environment variable that reports back the name of the local computer:

```
[atl-ocs-001]: PS C:\Users\Administrator\Documents> $env:computername
```

When we run this command we should get back the following:

```
atl-ocs-001

[atl-ocs-001]: PS C:\Users\Administrator\Documents>
```

Notice the prompt? As you can see, even though we just ran a command, we're still working on the remote computer atl-ocs-001; that's because we have a persistent connection. Let's try another command and see what happens:

```
[atl-ocs-001]: PS C:\Users\Administrator\Documents> Get-Process

Handles  NPM(K)    PM(K)     WS(K) VM(M)   CPU(s)      Id  ProcessName
-------  ------    -----     ----- -----   ------      --  -----------
     47       5     1852      4352    62     0.08     340  notepad
     39       5     1852      4332    43     0.11    1128  calc
     87       5     1848      4288    118      0.08   1260  winword

[atl-ocs-001]: PS C:\Users\Administrator\Documents>
```

Again, take a look at the prompt: we're *still* working on the remote computer atl-ocs-001. And that will last until the end of time or until we decide to terminate our session.

Whichever comes first.

Speaking of which, what if we *have* decided to terminate our session? That's fine; all we have to do is type the **exit** command:

```
[atl-ocs-001]: PS C:\Users\Administrator\Documents> exit

PS C:\Scripts
```

As you can see (hint: look at the prompt) we're no longer working on the computer atl-ocs-001. Instead, we're back on our local computer, and back to the good old C:\Scripts folder to boot. You know what they say: be it ever so humble, there's no place like home.
And yes, now that you mention it, we *can* create a remote session that runs on multiple computers at the same time. In fact, you already know how to do that, don't you?

```
PS C:\Scripts> Enter-PSSession -ComputerName atl-ocs-001,atl-ocs-002,`
    atl-ocs-003
```

You can also take a different tack: you can open a session to computer A, open a separate session to computer B, and then switch back and forth between those sessions. Take a look at these commands:

```
PS C:\Scripts> $a = New-PSSession -ComputerName atl-ocs-001
PS C:\Scripts> $b = New-PSSession -ComputerName atl-ocs-002
PS C:\Scripts>
```

As you can see, in the first command we used the **New-PSSession** cmdlet to open a new PowerShell session on the remote computer atl-ocs-001; that session was then saved in a variable named $a. In the second command, we opened a new PowerShell session on the remote computer atl-ocs-002; that session was saved in a variable named $b. And after we execute that second command we – oh, looks like we're still on the local computer, and still in the C:\Scripts folder, aren't we?Apparently our two new commands didn't work, did they?

Oh ye of little faith. Our commands *did* work. New-PSSession did just what it was supposed to do: it created two new PowerShell sessions. We can verify that by using **Get-PSSession** to list the remote sessions currently running on our computer:

```
Id   Name       ComputerName    State    Configuration
 --  ----       ------------    -----    -------------
  1  Session1   atl-ocs-001     Opened   Microsoft.PowerShell
  2  Session2   atl-ocs-002     Opened   Microsoft.PowerShell
```

As you can see, we have two open connections. The only reason our prompt still reflects the local computer is because all we've done so far is open those connections; we haven't actually *entered* those connections yet. To do that, we need to call the Enter-PSSession cmdlet followed by the appropriate variable. For example, to enter session 2 (the one running on atl-ocs-002) we'd use this command:

```
PS C:\scripts> Enter-PSSession $b

[atl-ocs-002]: PS C:\Users\Administrator\Documents>
```

As usual, look at the prompt: we're now working on the computer atl-ocs-002.

Now, suppose we want to exit this session. As you might expect, all you have to do is type the **exit** command:

```
[atl-ocs-002]: PS C:\Users\Administrator\Documents> exit

PS C:\Scripts>
```

From this point we could now enter session 1 (**Enter-PSSession $a**), or we could re-enter session 2. In fact we – what's that? You know, now that you mention it, session 2 *is* still running, isn't it? Take a look at the results of running Get-PSSession:

```
 Id  Name       ComputerName    State    Configuration
 --  ----       ------------    -----    -------------
  1  Session1   atl-ocs-001     Opened   Microsoft.PowerShell
  2  Session2   atl-ocs-002     Opened   Microsoft.PowerShell
```

But, you ask, didn't we just run the **exit** command? Yes, we did, and PowerShell dutifully exited the session. However, when you are running multiple PowerShell sessions exiting does *not* terminate the connection; it merely returns you back to the local computer. If you truly want to terminate the connection you need to exit and then run the **Remove-PSSession** cmdlet:

```
PS C:\Scripts> Remove-PSSession $b
```

Now look at what we get back when we run Get-PSSession:

```
Id   Name       ComputerName    State    Configuration
 --  ----       ------------    -----    -------------
  1  Session1   atl-ocs-001     Opened   Microsoft.PowerShell
  2  Session2   atl-ocs-002     Closed   Microsoft.PowerShell
```

Session 2 is now shown as closed. And, trust us, it's really closed. Look what happens if we try to re-enter the session:

```
PS C:\Scripts> Enter-PSSession $b

Enter-PSSession : Session must be open.
At line:1 char:16
+ Enter-PSSession <<<<  $b
    + CategoryInfo          : InvalidArgument: (:) [Enter-PSSession],
ArgumentException
    + FullyQualifiedErrorId : PushedRunspaceMustBeOpen,Microsoft.PowerShell.Commands.
EnterPSSessionCommand
```

We can't enter the session, because the session is no longer open.

Note. Although the session is closed, $b will still contain an object reference to that session. To remove that object reference you can simply delete the variable:

```
Remove-Variable b
```

"When you are running multiple PowerShell session exit-ing does not terminate the connection; it merely returns you back to the local computer. "