



realtimepublishers.com<sup>tm</sup>

# *The eBooklet Series*<sup>tm</sup>

## **An Introduction to: Microsoft<sup>®</sup> PowerShell<sup>™</sup>**

*Don Jones*

## Introduction to Realtimepublishers

by Don Jones, Series Editor

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

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

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

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

Until then, enjoy.

Don Jones

Introduction to Realtimerepublishers.....	i
An Introduction to Microsoft PowerShell.....	1
If You Don't Know History... You're Doomed to Repeat It.....	1
So What Is the Solution?.....	1
What Is PowerShell and Why Should I Care?.....	2
PowerShell Requirements.....	3
Quick Start.....	3
Navigating Your System.....	4
Using the PowerShell Command Line.....	5
Aliases.....	5
Basic Cmdlets.....	6
Parameters.....	7
Ubiquitous Parameters.....	7
Profiles.....	7
Scripts.....	8
Redirection and Substitution.....	8
Variables.....	9
Special Characters.....	13
Scopes.....	14
Functions.....	15
Pipelines.....	15
Getting Help.....	16
So What Has .NET Got to Do With It?.....	16
Microsoft .NET Framework Essentials.....	17
Reflection.....	18
Assemblies.....	18
Variables as Objects.....	18
The .NET Application Programming Interface in PowerShell.....	20
Advanced .NET in PowerShell.....	23
Okay—When Can I Get it? What Can it Do?.....	23
Want to Know More?.....	24

## Copyright Statement

© 2006 Realtimerepublishers.com, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtimerepublishers.com, Inc. (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtimerepublishers.com, Inc or its web site sponsors. In no event shall Realtimerepublishers.com, Inc. or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtimerepublishers.com and the Realtimerepublishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.

If you have any questions about these terms, or if you would like information about licensing materials from Realtimerepublishers.com, please contact us via e-mail at [info@realtimerepublishers.com](mailto:info@realtimerepublishers.com).

---

[**Editor's Note:** This eBook was downloaded from Realtime Nexus—The Digital Library. All leading technology guides from Realtimepublishers can be found at <http://nexus.realtimepublishers.com>.]

## An Introduction to Microsoft PowerShell

For ages, Microsoft has been blamed for being a bit lacking when it comes to scripting and automation for its flagship Windows product and related server products, such as Exchange Server and SQL Server. In the beginning, Windows NT simply included an MS-DOS-like command shell called Cmd.exe; in the mid-nineties, Microsoft introduced Visual Basic Scripting Edition—VBScript—as an administrative tool. In the interim, dozens of third-party or unsupported scripting solutions debuted with varying degrees of success: KiXtart, WinBatch, WinScript, AutoIt, and more, just to name a few. They all shared a single problem: Windows itself.

### If You Don't Know History...You're Doomed to Repeat It

Understanding the problems with Windows automation can help prevent those problems in the future, and the main problem—simply put—is *windows*. With a lowercase “w.” Windows (the product) has been entirely built around a graphical user interface (GUI), and how exactly do you automate something like that? You can't. Instead, early automation was provided in the form of command-line tools, which were really an *alternative* to the GUI; they rarely did the exact same thing as the GUI because they were built for different purposes. That meant the only things you could really script and automate were the things Microsoft offered command-line tools for—small wonder, then, that the tool-packed Windows NT Resource Kit was a bestseller!

VBScript tried to address some of these issues by allowing administrators to write scripts that more directly addressed the Windows operating system (OS) or, more specifically, the Component Object Model (COM) on which Windows is built. Most GUI administrative tools use COM to do their work, so by tapping directly into COM—this is the theory, at least—you could do anything the GUI could do. Or not; turns out VBScript couldn't address *every* COM interface, and some significant portions of Windows remained outside the realm of scriptability.

Another problem—both with command-line tools and with VBScript—was *consistency*. Every command-line tool and COM interface worked slightly differently, creating a learning curve every time you needed to use a new one.

### So What Is the Solution?

The solution was easy: Like UNIX/Linux (\*nix, for short) systems, Windows needed to be built upon a common ground that both GUI and command-line (scripting) administrative tools could access equally. Fortunately, by the early 21<sup>st</sup> century, that common ground existed: The Microsoft .NET Framework. What Microsoft needed was to *commit* to .NET and build *all* administrative functionality on it—something they did with Exchange Server 12 in 2006. Then, a powerful new command-line shell—one similar to Cmd.exe in certain ways and to VBScript in others—could be created that also interfaced with the .NET Framework, providing a simplified way of having the Framework perform administrative tasks. The answer was code-named “Monad,” and in early 2006 Microsoft officially announced it as PowerShell.

---

## What Is PowerShell and Why Should I Care?

Administrators of \*nix systems have always had the luxury of administrative scripting. In fact, most \*nix OSs are built on a command-line interface (CLI); while most also feature a GUI, the real work is done from the CLI. Every variant of \*nix supports some sort of shell scripting language that enables CLI commands to be strung together to automate administrative tasks.

Microsoft Windows has traditionally been built on a GUI rather than on a CLI; the exact opposite, in fact, of a typical \*nix system. Automating tasks performed in a GUI is significantly more difficult than automating tasks performed in a CLI; how do you write a script, for example, that tells a computer to select a certain check box if the contents of a text box are such-and-such? You really can't. To help administrators automate various tasks, Microsoft has traditionally included a variety of CLI tools—command-line *executables*—that provide a CLI-based way of performing tasks; by stringing these commands together in *batch* files, or scripts, administrators could automate these tasks. However, the CLI tools typically only exposed a *portion* of Windows' functionality, meaning you could only automate tasks for which Microsoft provided CLI tools.

In the late nineties, Microsoft introduced Visual Basic Scripting Edition, commonly referred to as VBScript. This scripting language was compatible with Microsoft's COM, which forms the building blocks of Windows itself. Because most GUI administrative tools were built on, and utilized, COM, it was felt that VBScript would be able to provide a better automation environment. Unfortunately, VBScript can still only automate a fraction of Windows' capabilities, although it can do far more than the simple CLI batch language (which evolved from Microsoft's earliest OS, MS-DOS).

Both CLI tools and VBScript have other problems, primarily in consistency. Because both evolved over time and were created by various groups within Microsoft who had no standards to work from, each CLI tool and COM interface (as used by VBScript) works a bit differently. Thus, every new tool or COM interface has a new learning curve, which takes additional time—which you may not have. All of this stems from the fact the Microsoft never really committed to scripting and automation for Windows; the feeling was that it was *Windows* and you primarily used the GUI to run it. As Windows' penetration into large companies and enterprises increased, however, managers and administrators accustomed to \*nix began to demand the same scripting and automation capabilities from Windows.

Which brings us to PowerShell. PowerShell is Microsoft's first comprehensive, from-scratch effort to create a scriptable automation shell for Windows. It's built on the Microsoft .NET Framework, which has deep ties into almost every aspect of the OS. Because Microsoft's made a strategic commitment to .NET, PowerShell's future is fairly secure because it will be built on the same platform that most of the rest of Microsoft's products will be built on. And PowerShell is, above all else, *consistent*: There are clear guidelines for how PowerShell is to be built and extended, meaning you won't have to learn an entirely new way of doing things every time you start a new script.

---

Exchange Server 12 is perhaps the best example of how PowerShell can be leveraged. PowerShell was built into this version of Exchange from the outset. In fact, *all* of the product's administrative functionality was built in .NET and exposed through PowerShell; the administrative GUI, or console, simply utilizes that underlying functionality. That means *any* Exchange administrative task can be performed in PowerShell—which means *any* task can be scripted and automated in a consistent fashion. Whether future use of PowerShell is so comprehensive remains up to the individual product groups within Microsoft, but with the strategic commitment both to .NET and to administrative automation, you can bet that PowerShell will finally offer a clear, consistent, and comprehensive tool for Windows administrative scripting.

### **PowerShell Requirements**

PowerShell is designed to run on all recent versions of Windows, including those based on x64 processors. The only prerequisite for installing PowerShell is that you must first install v2.0 of the Microsoft .NET Framework.



PowerShell will pre-install in certain situations; it is part of the Exchange Server 12 administrative tools, for example.



This eBooklet is based on Beta 3.1 of PowerShell.

### **Quick Start**

PowerShell is easy to get up and running—simply run PSH.EXE (or select the Start menu shortcut) and you'll be in the new shell. PowerShell is a complete shell, not unlike the Cmd.exe shell you're probably already familiar with. From within PowerShell, you can run normal applications such as Notepad or Calc; for applications that produce textual output (as opposed to using a GUI), you can capture the output within the PowerShell shell.

Under Cmd.exe, you typically ran CLI utilities such as Dir, Xcopy, Cacls, and so forth; under PowerShell, you'll primarily work with *cmdlets* (pronounced, “command lets”). Cmdlets serve the same role within PowerShell as CLI tools did under Cmd.exe, but they're all built to a consistent standard, and they're all built using the .NET Framework. PowerShell scripting involves stringing these cmdlets together to perform various tasks; if you're a .NET developer, you can also write your own cmdlets.

Cmdlets are always named in a *verb-noun* format, such as Get-Process. You can use the built-in Get-Help cmdlet to read help, when available for other cmdlets: Get-Help Get-Process, for example, displays help on the Get-Process cmdlet.

## Navigating Your System

The old Cmd.exe shell primarily provided access to drives, files, and folders on your system; PowerShell provides access to these as well as to additional resources such as the Windows registry. However, PowerShell “maps” these additional resources so that they look like drives, providing a familiar interface for working with a variety of resources. For example, when you open PowerShell, you might have a prompt that looks something like this:

```
PSH C:\>
```


that indicates that PowerShell is currently looking at the root of the C drive on your system. You can see a list of current drive mappings by using the Get-Drive cmdlet:

Name	Provider	Root	CurrentLocation
A	Microsoft....	A:\	
Alias	Microsoft....		
C	Microsoft....	C:\	...TEM\MSMAPI\1033
cert	Microsoft....	\	
D	Microsoft....	D:\	
Env	Microsoft....		
F	Microsoft....	F:\	
Function	Microsoft....		
HKCU	Microsoft....	HKEY_CURRENT_USER	
HKLM	Microsoft....	HKEY_LOCAL_MACHINE	
Variable	Microsoft....		

Notice that drive names aren't limited to single letters: The HKLM drive, for example, maps to the HKEY\_LOCAL\_MACHINE portion of the registry. Also notice the Provider column, which indicates the PowerShell provider, or piece of software, that is providing the connectivity to that particular resource. Providers are what make PowerShell so flexible. Simply by adding a provider, you can gain access to entirely new resources through PowerShell. PowerShell ships with providers that give you access to certificates (the cert provider), the registry (HKCU and HKLM), the file system (drive letters), and the Windows environment (the env provider), in addition to internal providers for aliases, functions, and variables.

To change locations, simply use the Set-Location cmdlet, passing it the name of the location you want to change to:

```
PSH C:\>Set-Location HKLM:\
```

 PowerShell is generally case-insensitive, so Set-Location is the same as set-location.

Note that you can set yourself directly to a complete location:

```
PSH C:\>Set-location "C:\Documents and Settings"
```

This command will change directly to the indicated path. Note that the path is contained within double quotation marks because it contains spaces; any argument that contains spaces *must* be enclosed within double quotation marks.



---

PowerShell also maintains a *stack* of locations. You can add, or *push*, a location onto the stack by using the Push-Location cmdlet; you can quickly change to the location on the top of the stack by using the Pop-Location cmdlet. For example:

```
PSH C:\>Push-Location C:\
```

moves the location C:\ to the top of the stack. Later, when you're ready to quickly change back to C:\, just issue:

```
PSH C:\Documents and Settings>Pop-Location
```

Learning to navigate through the PowerShell shell quickly is a key to using it effectively.

### **Using the PowerShell Command Line**

PowerShell has some very basic line-editing capabilities that you can use when typing at the command line. These are no substitute by any stretch for a full development environment if you're writing scripts or cmdlets, but they provide basic features when you just need to run a script or cmdlet interactively:

- Down- and up-arrow displays previously entered commands
- Left- and right-arrow move the cursor left and right, respectively
- The Home key moves to the beginning of the current command; End moves to the end
- Ctrl+Left and Ctrl+Right jump one word to the left and right
- Insert toggles insert/overwrite mode
- Backspace deletes the character in front of (to the left of) your cursor; Delete removes the character to the right of your cursor
- Press Tab to auto-complete path names

### **Aliases**

As intuitive as PowerShell's cmdlet names can be, they're not always convenient to type. Typing Set-Location is a poor substitute for the good ol' cd command under Cmd.exe, for example. Thus, PowerShell lets you define *aliases*, or nicknames, for cmdlets. For example, if you find Pop-Location to be too cumbersome, create a nickname called "Popd" for it:

```
PSH C:\>Set-Alias popd Pop-Location
```

And now you can use Popd in place of Pop-Location. You can remove, or un-define, an alias by using the generic Remove-Item cmdlet:

```
PSH C:\>Remove-Item alias:popd
```

This command removes the Popd alias from the system. PowerShell predefines a number of useful aliases; just run Get-Alias to see them all. Here's a portion of the output you'll see:

CommandType	Name	Definition
-----	----	-----
Alias	ac	add-content
Alias	clc	clear-content
Alias	cli	clear-item
Alias	clp	clear-property
Alias	clv	clear-variable
Alias	cpi	copy-item
Alias	cpp	copy-property
Alias	cvpa	convert-path
Alias	epal	export-alias
Alias	epcsv	export-csv
Alias	gci	get-childitem

Note that you can only set up aliases for cmdlets; aliases aren't shortcuts for entire command strings. For example, this won't work:

```
PSH C:\>Set-Alias GoC "Set-Location C:\"
```

Aliases can *only* be for a single cmdlet or external executable, not for any accompanying parameters or arguments.

### Basic Cmdlets

PowerShell will happily provide a list of all registered cmdlets, using Get-Command. Here's a partial list:

CommandType	Name	Definition
-----	----	-----
Cmdlet	add-content	add-content [-Path] String[...]
Cmdlet	add-history	add-history [[-InputObject] ...]
Cmdlet	add-member	add-member [-Type] MshMember...
Cmdlet	add-mshsnapin	add-mshsnapin [-Name] String...
Cmdlet	clear-content	clear-content [-Path] String...
Cmdlet	clear-item	clear-item [-Path] String[...]
Cmdlet	clear-property	clear-property [-Path] Strin...
Cmdlet	clear-variable	clear-variable [-Name] Strin...
Cmdlet	combine-path	combine-path [-Path] String[...]
Cmdlet	compare-object	compare-object [-ReferenceOb...]
Cmdlet	convert-HTML	convert-HTML [[-Property] Ob...]
Cmdlet	convert-path	convert-path [-Path] String[...]

That's a good way for inventorying what your capabilities within PowerShell are. For any given cmdlet, Get-Command will tell you more about it. For example:

```
PSH C:\>Get-Command Set-Alias
```

will describe what the Set-Alias cmdlet does. You can also use wildcards:

```
PSH C:\>Get-Command Get-*
```

And you can use Get-Help to learn more specific information. For example:

```
PSH C:\>Get-Help Set-Alias
```

produces a lengthy description of the Set-Alias cmdlet, along with details about each argument, examples of how the cmdlet is used, and so forth.

---

## Parameters

Many cmdlets can accept parameters; these are passed by name, using a hyphen, then the parameter (or argument) name, a space, and then the value you're passing to the parameter. For example:

```
PSH C:\>New_Item -type file "myfile.txt"
```

will create a new file. Notice the `-type` parameter, which is given the values `file` and a filename. When parameters are passed by name, they can be passed in any order. Some parameter names may be abbreviated, such as `-db` for `-debug`; valid abbreviations are always listed in the command's help.

## Ubiquitous Parameters

Most cmdlets support a set of *ubiquitous parameters*, which are always optional (meaning they don't need to be specified if you don't want to use them):

- `-Debug (-db)`—Instructs the cmdlet to provide additional programmer-level detail about the operation.
- `-ErrorAction (-ea)`—Controls the behavior of the cmdlet when an error occurs. Values can be `NotifyContinue` (which is the default), `NotifyStop`, `SilentContinue`, `SilentStop`, and `Inquire`.
- `-ErrorVariable (-ev)`—Specifies the name of a variable in which to place all objects to which an error occurred while processing. The specified variable is processed in addition to the built-in `$ERROR` variable.
- `-OutVariable (-ov)`—Specifies the name of a variable in which to place all objects that are output from the cmdlet.
- `-Verbose (-vb)`—Instructs the cmdlet to produce additional output about its actions and progress.

## Profiles

PowerShell uses profiles to help customize the shell environment. There are a few files that customize the profile:

- `\Documents and Settings\All Users\Documents\PSH\profile.PSH`
- `\Documents and Settings\All Users\Documents\PSH\Microsoft.Management.Automation.PSH_profile.PSH`
- `\My Documents\PSH\profile.PSH`
- `\My Documents\PSH\Microsoft.Management.Automation.PSH_profile.PSH`

These files, if they exist, are read in this order; conflicts are “won” by whichever file is read last. If none of these files exist, PowerShell will use its built-in default settings.

---


## Scripts

So far, we've simply explored ways to run cmdlets directly and view their output. PowerShell scripts are designed to string several cmdlets together to automate more complex tasks.

PowerShell scripts must have the filename extension .PSH; to run a script, simply type its name, without the extension. PowerShell will look in the environment Path variable or any files with the .PSH extension in order to find your script. If your script takes input arguments, simply type them after the script name:

```
. Myscript arg1 arg2
```

Did you catch the period at the beginning of the line? That tells PowerShell to execute the script in the current *scope*, which I'll discuss in a bit. Scripts use their own language, which is similar to both VBScript and C#, and I'll spend most of the rest of this guide discussing this scripting language.

 How can PowerShell tell when you're typing a script name, a cmdlet name, or an alias? When you type *any* name, PowerShell tries to look first for an alias, then a function, then a cmdlet, then a script, and then an external executable.


Why did Microsoft choose to use a new language rather than a language already in existence, like VBScript? A few reasons come to mind. First, because PowerShell is built on .NET, the scripting language needed to be able to leverage .NET's features and capabilities, which VBScript certainly can't do. In fact, no scripting language had existed that could really utilize .NET. A new language could also be more consistent than languages like KiXtart, which evolved over time and are a bit of a mishmash. Microsoft decided to go with a language that was essentially a subset of the C# .NET language, allowing an easier "upscale" from PowerShell to the full C# language, should you ever want to make that leap.

## Redirection and Substitution

One common thing you'll need to do is redirect the output of one cmdlet into another cmdlet, tying the two together. You may also want to redirect cmdlet output to a file, in order to create a report of some kind. For example, to create your own reference file of available cmdlets:

```
PSH C:\>Get-Command > commandref.txt
```

You'll have a file named Commandref.txt, located in the current location (as indicated by the PowerShell prompt), which contains the output of the Get-Command cmdlet.

 To append output to an existing file, rather than overwriting it, use >> instead of >.

You can use the output of one cmdlet as the input, or argument, to another cmdlet or language expression. The syntax is  $\$(cmdlet)$ , as in this example:

```
PSH C:\>Get-ChildItem $(Read-Host -Prompt "Enter filename: ")
```

---

This idea is a bit difficult to follow, so let's walk through it slowly: The first cmdlet is `Get-ChildItem`. This cmdlet is designed to accept a file path, then display the child items—files and subfolders, usually—of that path. The `Read-Host` cmdlet is designed to read input from the command line; its `-Prompt` argument defines a text prompt. Thus, this example displays the following:

```
PSH C:\> Get-ChildItem $(Read-Host -Prompt "Enter filename: ")
Enter filename: : C:\

    Directory: Microsoft.Management.Automation.Core\FileSystem::C:\

Mode                LastWriteTime         Length Name
----                -
-a----            1/10/2005   9:01 PM         15320 ArchiveLogs.wsf
-a----            1/9/2005   10:07 PM           0 AUTOEXEC.BAT
-a----            4/10/2006   10:12 AM          17 computers.txt
-a----            1/9/2005   10:07 PM           0 CONFIG.SYS
-a----            4/12/2006   11:47 AM          526 hpfr5550.xml
d-----            2/26/2006    5:21 PM                Documents and Settings
d-----            1/9/2005   10:32 PM                Inetpub
d-----            3/2/2006    1:29 PM                logs
d-----            4/17/2006   11:34 AM                Program Files
d-----            4/13/2006    1:31 PM                temp
d-----            4/13/2006    8:56 PM                WINDOWS
```

The output of `Read-Host`—that is, whatever was typed at the prompt—is passed as the input argument to `Get-ChildItem`.

Other forms of substitution are possible. For example, to create five files named 1, 2, 3, 4, and 5:

```
PSH C:\>New_Item -type file $(1..5)
```

The `New-Item` cmdlets has an input argument, `-Type`, which accepts an item type (I've specified file) and name. For the name, I've used substitution, specifying that the values 1 through 5, inclusive, should be used. This causes the cmdlet to run once for each value I've supplied, creating five new files.

## Variables

Like many scripting environments, PowerShell supports the creation and use of *variables*. Think of a variable as a container that has a name and can hold values. For example:

```
PSH C:\>$var = 100
```

creates a new variable named `$var` and assigns it the numeric value 100. Variable names always begin with `$` in PowerShell.

---

You can declare variables right at the PowerShell prompt; you don't need to be running a script:

```
PSH C:\Documents and Settings> $var = "C:\"
PSH C:\Documents and Settings> set-location $var
PSH C:\>
```

In the first line of this example, the variable `$var` is declared and set to contain the string value `"C:\"`—notice the double quotation marks around the value, which marks it as a string rather than a number. The second line executes the `Set-Location` cmdlet, passing the contents of `$var`. As you can see from the prompt on the third line, the location was successfully changed to `C:\`. It's important to note that, when variables are used, it's the *contents* of the variable that are passed along, not the variable name itself. In other words, I wasn't trying to set the location to a location named `"$var;"` I set it to whatever was *contained within* the variable `$var`.

Variables can contain the output of cmdlets, too:

```
PSH C:\>$a = get-process
```

will run the `Get-Process` cmdlet and put its entire output into the variable `$a`.

## Variable Names and Intrinsic Variables

Variable names can contain *any* character. However, if they don't start with a letter, they must be enclosed in curly braces:

```
$var = 4
$var2 = 3
${@@123} = 2
```

It can be a bit confusing to use variable names such as `@@123`, so I recommend sticking with textual, meaningful names. Interestingly, a variable name can be a path, such as:

```
PSH C:\>${C:\File.txt} = "Hello!"
```

This command will write `"Hello!"` to a text file named `C:\File.txt`. Remember that *every resource* to which PowerShell connects can be presented with a file-like path—the path `HKLM\SOFTWARE`, for example, goes to the registry—so this can be a powerful technique for quickly changing values in various resources.

PowerShell provides a number of built-in variables—automatic variables, policy variables, and so forth—that provide information about the current environment, the currently executing host, and so forth. These are listed in the PowerShell documentation, and you shouldn't try to name your own variables any of the names used by these built-in variables.

---

## Variables Are Objects

It's important to understand that PowerShell variables are objects. This is unlike languages such as VBScript, where variables are simply containers for values; in the case of PowerShell, a variable *does* contain a value, but it also has a number of intrinsic capabilities because it's also an object. For example:

```
PSH C:\>$var = "Hello, World"
```

assigns the value "Hello, World" to the variable \$var. \$var now contains that variable, but \$var also has a number of capabilities as an object. One of those capabilities is the SubString() method:

```
PSH C:\>Write-Host $var.SubString(2,2)
```

This command calls the Write-Host cmdlet, which outputs text to the console. It asks that the \$var object execute its SubString() method, which starts at the third character position (numbering begins with 0, so 2 is the third character) and takes 2 characters. This will output "ll" to the console. Similarly:

```
PSH C:\>Write-Host $var.Length
```

would output the number 12, because that's how long the contents of \$var are: 12 characters. Note that PowerShell doesn't visually differentiate between variables that contain strings and those that contain numbers:

```
$var = 5
```

```
$var = "Hello"
```

Both are perfectly legal. However, PowerShell *can* tell the difference. Variables containing string values are referred to as *string objects*, and they come with a rich variety of methods and properties such as SubString() and Length. For example:

```
PSH C:\> $var = 3
```

```
PSH C:\> write-host $var.length
```

```
PSH C:\> $var = "Hello"
```

```
PSH C:\> write-host $var.length
```

```
5
```

First, \$var is given the numeric value 3. When asked to output the length, PowerShell can't because 3 isn't a string; thus, \$var isn't a string object. However, when the contents of \$var are replaced by the string "Hello," \$var becomes a string object and has a valid Length property, as shown in the output.

---

## String Variables and Embedding

String variables treat embedded variables in an interesting fashion. For example:

```
PSH C:\> $var = "Hello"
PSH C:\> $var2 = "$var, World!"
PSH C:\> write-host $var2
Hello, World!
```

In this example, the value “Hello” was assigned to \$var. The value “\$var, World!” was assigned to \$var2. When passed to Write-Host, \$var was expanded, meaning its *contents* were displayed. That is because \$var2 was assigned using double quotation marks. Now, consider this similar example:

```
PSH C:\> $var = "Hello"
PSH C:\> $var2 = '$var, World!'
PSH C:\> write-host $var2
$var, World!
```

Notice the difference? This time, the value passed into \$var2 was contained in *single quotation marks* instead of double. This prevented \$var from being expanded, and so the literal value “\$var, World!” was stored in \$var2, as evidence by the Write-Host output. \$var2 is still considered a string object; either single quotes or double quotes can be used to contain strings.

Strings assigned with double quotation marks can also contain embedded expressions. For example:

```
PSH C:\> $var = "2+2 is $(2+2)"
PSH C:\> write-host $var
2+2 is 4
```

Anything with a \$ is considered either a variable or expression and is evaluated accordingly. In this case, the expression (2+2) was recognized as a mathematical expression, and it was evaluated for its result. Here’s one last useful example:

```
PSH C:\> $var = "Hello"
PSH C:\> $var2 = "$var, World!"
PSH C:\> $var = "Goodbye"
PSH C:\> write-host $var2
Hello, World!
```

Notice that the output of Write-Host is “Hello, World!” and not “Goodbye, World!” as you might expect. The reason is that \$var was expanded *when it was assigned into \$var2*. In other words, \$var2 contains the static string, “Hello, World!” because \$var contained “Hello” at the time the value was assigned to \$var2. Later changes to \$var do not effect the existing contents of \$var2.



---

## Parsing Mode

All of this quotation stuff can get confusing because it works somewhat differently at the command line, when you're just typing text into PowerShell. For example:

```
PSH C:\> write-host 2+2
2+2
PSH C:\>
```

Why didn't it display 4? Because at the command line, everything is considered a string unless it appears in parentheses or starts with \$ (meaning it's a variable). Thus, this works differently:

```
PSH C:\> write-host (2+2)
4
PSH C:\>
```

Why the difference? At the command line, PowerShell treats everything as a string so that you don't have to put quotation marks around everything. That allows you to run:

```
PSH C:\>Set-Location C:\
```

Rather than having to type:

```
PSH C:\>Set-Location "C:\"
```

Which would be cumbersome and unintuitive because it's not the way past Windows shells have worked. This is all called the shell *parsing mode*: Whether the shell treats things as strings by default, or not. The rules are pretty simple: If the first character is a number, a variable (\$), or a quoted string, the shell works in *expression mode*, in which all strings must be quoted. If the first character is a letter, ampersand (&), or a dot followed by a space or a letter, the shell works in *command mode*, which is where everything is assumed to be a string unless it's a variable or is in parentheses, as I've demonstrated.

## Special Characters

Sometimes, you might need to display special characters that can't be typed. PowerShell provides an *escape character* for these: ```, which is a backward apostrophe, usually located on the same key as `~` on your keyboard. The special characters are:

Character	Escape Code
Null	<code>`0</code>
Alert	<code>`a</code>
Backspace	<code>`b</code>
Form feed	<code>`f</code>
New line	<code>`n</code>
Carriage return	<code>`r</code>
Tab	<code>`t</code>
Vertical quote	<code>`v</code>

In order to display a ``` by itself, type ````.

---

## Scopes

*Scope* is a description of the visibility of a function or variable within PowerShell. This is a means of controlling access to variables and functions. Generally, unless you explicitly request otherwise, variables can be read and changed only within the scope where they were originally created, and they'll only be accessible to cmdlets running in the same scope. That's why, in the previous example of running a script:

```
PSH C:\>. Myscript arg1 arg2
```

It was so important to specify the period, to ensure that the script would run in the *current* scope—thus having access to any variables or functions already declared within that scope. If I hadn't used the period, PowerShell would have taken its default action of creating a new scope for the script. This technique of preceding the script name with a period and a space is called *dot sourcing*, and it essentially makes the script behave as if each line of the script was being typed, by you, right into the PowerShell shell.

All scripts, by default (that is, unless you use dot sourcing), are run in a newly created scope. Child scopes—that is, scopes created by another scope—can *read* variables from the parent scope but not *change* them as easily. Parent scopes can't access child scope variables in any way.

When you start a new instance of PowerShell, you're working in the *global* scope. Any child scope can access global scope variables (such as environment variables), but they must explicitly label the variable as global in order to do so (I'll touch more on this later).



The global scope is simply named global, while the scope of an executing script is named script. I'll reveal other special scope names as I use them elsewhere in the book.

For example, suppose a script declares a variable named \$var. A function runs, and also declares \$var. There are now two copies of \$var in existence—the one in the script's scope, and the one in the function's scope. Because the function is contained within the script, its scope is a child of the script's scope. Thus, the function *can* access script-level variables if it chooses to do so, by referring to \$script:var—that is, the name of the scope (script) and the name of the variable from that scope (var). More on functions is coming up next.

Variables can also be declared as *private*, meaning they're accessible *only* from the current scope, and not from within child scopes:

```
PSH C:\>$private:var
```

Declares a private variable named \$var.

---

## Functions

Functions are little subroutines of code that are intended to be self-contained. Functions have their own scope, as outlined in the previous discussion on scope—variables declared within a function are accessible only to the function. If the function was run as part of a script, the function can access the script’s scope because the script is the parent of the function.

Functions are declared with the keyword `function`, given a name, and then can include whatever code you need. For example:

```
function myFunction {  
    $var = 3  
    $script:var = "Hello"  
}
```

Notice that the function’s code is enclosed by {curly braces}, which lets PowerShell know where the function starts and stops.

## Pipelines

One of the most powerful and possibly confusing aspects of PowerShell is its *data pipelines*. These provide a means of passing data and objects from one cmdlet to another in a very robust fashion. Perhaps you’ve used the `Cmd.exe` `More` utility to slow the display of a long directory:

```
C:\>Dir | More
```

This command takes the output of `Dir` and “pipes” it to `More`, which displays the data in nice pages, and waits for you to hit a key before displaying the next page. Pipes have the same basic function in PowerShell (in fact, that character in between `Dir` and `More` is called the *pipe character*, and it’s usually on the backslash key of your keyboard); here’s a robust example:

```
PSH C:\>Get-Process | where { $_.handlecount -gt 400 } | Format-List
```

☞ PowerShell has a `More` command, too, if you want to pipe multi-screen output to it for one-page-at-a-time display.

---

This example is actually executing *three* cmdlets. The first, `Get-Process`, returns a list of all running processes. Each process is actually an object, of sorts, with various properties. These are all piped to `where`, which is an alias for the `Where-Object` cmdlet. Its job is to sort through a list of objects and pull out those that match some criteria; in this case, where their `handlecount` property is greater than (that's the `-gt` argument) 400. All of that is piped to the `Format-List` cmdlet, which creates a nice, pretty list of the results:

```
PSH C:\> Get-Process | where { $_.handlecount -gt 400 } | Format-List

ProcessName : csrss
Id          : 1080

ProcessName : explorer
Id          : 1952

ProcessName : Groove
Id          : 2656

ProcessName : inetinfo
Id          : 1524
```

The ability of pipes to pass data to other cmdlets, and the ability of PowerShell to deal with complex, structured objects (like the `Process` object) in a text interface, is part of what has people so excited about it.

### Getting Help

PowerShell has a fairly comprehensive built-in help system. To see a list of all available help topics, type `Help *`; for help with just a specific topic, run `Help topicname`, such as `Help about_Alias`. `Help`, by the way, is an alias for the `Get-Help` cmdlet. For help on absolutely everything, including cmdlets and aliases, just run `Help *`.

### So What Has .NET Got to Do With It?

You probably don't *feel* as if you need to know much about the .NET Framework. Or maybe you're just *hoping* that's the case. Well, almost: You certainly don't need to know *much* about it, and what you do need to know will be summarized in this short section. You do need to know a little bit, though, in order for a lot of what PowerShell does to make sense to you. I'll also explore some WMI essentials, because WMI is at the heart of so many administrative tasks that you'll automate through PowerShell.

---

## Microsoft .NET Framework Essentials

.NET is Microsoft's leading-edge software development framework. Traditional .NET development begins inside a development environment such as Microsoft Visual Studio, SAPIEN PrimalScript Enterprise, or even Windows Notepad. Applications are written in languages like VB.NET or C#. They're then compiled to a special language called MSIL, the Microsoft Intermediate Language. This is important, because it's different from how other things, like Visual Basic 6, compiled programs into a native, binary executable.

When you double-click a .NET executable, it doesn't actually run right away. That's because it contains MSIL, not actual binary code. Instead, Windows fires up the .NET Common Language Runtime, or CLR. *That* is what reads the MSIL and compiles it into executable, binary code that'll run on your system. This makes .NET applications inherently portable: They can (more or less) run on any platform for which a CLR is available. This is all a bit of an oversimplification, of course, but it's more than close enough for Windows administrative work.

The point of all this is that PowerShell is itself built in .NET, as are the cmdlets you'll be running. .NET is an *object-oriented* framework, which is a fancy way of saying it's kind of template-based. For example, all PowerShell cmdlets start out as copies of a standardized cmdlet *base class* or template. In programmer terms, you would say that all cmdlets *inherit from* that cmdlet base class. This is important because it's what makes all cmdlets pretty consistent with one another, allows them to all share certain ubiquitous parameters, and so forth.

The object-oriented stuff also plays heavily into how PowerShell works, but to understand why, you need to know a bit more about what an object *is*. At the simplest level—one suitable for cocktail parties, perhaps—an object is just a bunch of computer code all bundled up into a “black box.” The box has buttons you can push to make things happen, and has little blinky lights to tell you what's going on inside. You don't actually know how the box works—inside it could be anything from a particle accelerator to a cheese sandwich—and it doesn't really matter; the whole point is that you only interact with the box through its blinky lights and buttons, and everything inside is a big mystery. You can build your own black box, which incorporates another black box (this is called inheritance), essentially installing box number one inside box number two, so that your box (number 2) can take full advantage of number one's functionality without really know much about what goes on inside.

In .NET—and in PowerShell—*everything* is an object. Every variable you create, every WMI class you return, *everything* is an object. And those objects all have buttons—called *methods*—and blinky lights—called *properties*. For example, when you run:

```
PSH:> $stuff = get-wmiobject -class Win32_Process  
           namespace -root\cimv2
```

The Get-Wmiobject cmdlet goes out and gets all the instances of the Win32\_Process WMI class. Each class is an object. Together, they're bundled up into a *collection* of objects, which is stored in \$stuff. So the variable \$stuff is now a collection (or *list*, or *array*, if you prefer one of those terms instead) of Win32\_Process instances. Even a simple string of text “like this one” is really an object—a *string* object, to be specific—as far as PowerShell and .NET are concerned.

---

## Reflection

Part of what made Microsoft's COM (the pre-PowerShell way of managing Windows, often through a language like VBScript) so difficult is that objects (COM had objects, too) had to take special steps to define their functionality ahead of time. In other words, when someone at Microsoft created a DLL that allowed your scripts to work with files and folders, they also had to create a little file called a *type library* that explained what the DLL was capable of. That made COM difficult to extend easily. With .NET, however, there is a nifty feature called *reflection*. Basically, it's just a way for one application—such as PowerShell—to discover something about an object at runtime without having to be told in advance what the object can do. Reflect makes PowerShell infinitely extensible, because you can just add new cmdlets and PowerShell can more or less ask the cmdlets themselves what they do and how they work.

## Assemblies

In .NET, everything gets packaged, eventually, into an *assembly*. Cmdlets, for example, are assemblies. It's really just a fancy word for what I'd otherwise call an executable, or a DLL, or some other file-that-contains-executable-code. You'll find assemblies distributed with PowerShell in (by default) the shell's installation folder: `System.Management.Automation.Commands.Management.dll`, for example, is a file containing bunches of different cmdlets (one assembly can actually contain, or *implement*, multiple objects or *interfaces*—each one being a separate cmdlet, for example).

## Variables as Objects

Earlier, I mentioned that even variables are objects, and they are. In .NET, string variables in particular are extremely robust, and have a number of methods and properties. One of them is `Split`, a method that takes a string and creates an array (or list) out of it by breaking the list up on some character, like a comma or a space. Try this in PowerShell:

```
PSH:> "1,2,3,4".Split(",")
```

What you're telling PowerShell is “take this string, and execute its `Split` method. Use a comma for the method's input argument.” PowerShell does this, and the method returns an array of four elements, each element containing a number. PowerShell gets that array and displays it in a textual fashion, with one array element per line:

```
1  
2  
3  
4
```

There are other ways to use this. For example, PowerShell has a cmdlet called `Get-Member`, which displays the methods and variables associated with a given object instance. Thus, taking a string such as “Hello, World”—which, remember, is an instance of a `String` object—and piping it to the `Get-Member` cmdlet will display information about that `String` object:

```
PSH C:\> "Hello, World" | get-member

    TypeName: System.String

Name                MemberType          Definition
----                -
Clone               Method              System.Object Clone()
CompareTo           Method              System.Int32 CompareTo(Object obj)
Contains            Method              System.Boolean Contains(String value)
CopyTo              Method              System.Void CopyTo(Char[] destination, Int32 destinationIndex)
EndsWith           Method              System.Boolean EndsWith(String value)
Equals              Method              System.Boolean Equals(Object obj)
get_Chars           Method              System.Char get_Chars(Int32 index)
get_Length          Method              System.Int32 get_Length()
GetEnumerator        Method              System.CharEnumerator GetEnumerator()
GetHashCode         Method              System.Int32 GetHashCode()
GetType             Method              System.Type GetType()
GetTypeCode         Method              System.TypeCode GetTypeCode()
IndexOf             Method              System.Int32 IndexOf(Char value)
IndexOfAny          Method              System.Int32 IndexOfAny(Char[] values)
Insert              Method              System.String Insert(Int32 index, String value)
IsNormalized        Method              System.Boolean IsNormalized()
LastIndexOf         Method              System.Int32 LastIndexOf(Char value)
LastIndexOfAny     Method              System.Int32 LastIndexOfAny(Char[] values)
Normalize           Method              System.String Normalize()
PadLeft             Method              System.String PadLeft(Int32 width)
PadRight            Method              System.String PadRight(Int32 width)
Remove              Method              System.String Remove(Int32 index, Int32 count)
Replace             Method              System.String Replace(Char oldChar, Char newChar)
Split               Method              System.String[] Split()
StartsWith          Method              System.Boolean StartsWith(String value)
Substring           Method              System.String Substring(Int32 start)
ToCharArray         Method              System.Char[] ToCharArray()
ToLower             Method              System.String ToLower()
ToLowerInvariant    Method              System.String ToLowerInvariant()
ToString            Method              System.String ToString()
ToUpper            Method              System.String ToUpper()
ToUpperInvariant    Method              System.String ToUpperInvariant()
Trim                Method              System.String Trim()
TrimEnd             Method              System.String TrimEnd()
TrimStart           Method              System.String TrimStart()
Chars               ParameterizedProperty System.Char Chars(Int32 index)
Length              Property             System.Int32 Length {get;}
```

This output is truncated a bit to fit in this book, but you can see that it includes every method and property of the `String`, and correctly identifies “Hello, World” as a “`System.String`” type—that being the unique *type name* that describes what I informally call a `String` object. You can pipe nearly *anything* to `Get-Member` to learn more about that particular object and its capabilities.

---

## The .NET Application Programming Interface in PowerShell

The fact that PowerShell is built on and around .NET gives PowerShell tremendous power that isn't always obvious. For example, I already explained that any PowerShell variable can contain any type of data. Well, that's really because *all* types of data—strings, integers, dates, and so forth—are .NET classes, and they all inherit from the base class named Object. A PowerShell variable can actually contain anything that inherits from Object although, as in the previous example with a string, PowerShell can certainly tell the difference between different classes that inherit from Object.

You can actually force PowerShell to treat objects as a more specific type. For example, take a look at this sequence:

```
PSH C:\> $one = 5
PSH C:\> $two = "5"
PSH C:\> $one + $two
10
PSH C:\> $one = 5
PSH C:\> $two = "5"
PSH C:\> $one + $two
10
PSH C:\> $two + $one
55
```

In this example, I gave PowerShell two variables. One contained the number five, and the other contained the string character “5.” Might look the same to you, but it's a big difference to a computer! But I didn't specify what type of data they were, so PowerShell assumed they were both simply of the generic Object type, and decided it would figure out something more specific when they were actually used.

When I added \$one and \$two, or 5 + “5,” PowerShell looked and said, “aha, this is addition: The first character is definitely not a string because it wasn't in double quotes. The second one *was* in double quotes but... well, if I take the quotes away it looks like a number, so I'll add them.” And I correctly got ten as the result.

But when I added \$two and \$one—reversing the order—PowerShell had a different decision to make. “I see addition, but this first operand is clearly a string. The second one is a generic Object, so let's treat it like a string, too, and just concatenate the two.” And so I got the string “55,” which is just the first five tacked onto the second.

But what about:

```
PSH C:\> [int]$two + $one
10
```

Same order as the example that got “55,” but this time I specifically told PowerShell that the generic object in \$two was an Int, or integer, which is a type PowerShell knows about. So it used the same logic as in the first example, added the two, and came up with ten.



---

You can actually force PowerShell to try and treat anything as a specific type. For example:

```
PSH C:\> $int = [int]"5"
PSH C:\> $int | get-member

        TypeName: System.Int32

Name      MemberType Definition
-----
CompareTo Method      System.Int32 CompareTo(Int32 value), System.Int
Equals    Method      System.Boolean Equals(Object obj), System.Boole
GetHashCode Method     System.Int32 GetHashCode()
GetType   Method      System.Type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode()
ToString  Method      System.String ToString(), System.String ToStrin
```

Here, the value “5” would normally be either a String object or, at best, a generic Object. But by specifying the type [int], I forced PowerShell to try and convert “5” into an integer before storing it in the variable \$int. The conversion was successful, as you can see where I piped \$int to Get-Member, which revealed the object’s type: System.Int32. Of course, PowerShell isn’t a miracle worker: Force it to convert something that plainly doesn’t make sense and it’ll complain:

```
PSH C:\> $int = [int]"Hello"

Cannot convert "Hello" to "System.Int32". Error: "Input string
was
not in a correct format."
At line:1 char:13
+ $int = [int]" <<<< Hello"
```

Because “Hello” can’t sensibly be made into a number. This one’s even more fun:

```
PSH C:\> $xml = [xml]"<users><user name='joe' /></users>"
PSH C:\> $xml.users.user

name
----
joe
```

I created a string, but told PowerShell that it was of the type XML, which is another data type PowerShell knows about. XML data works sort of like an object: I defined a parent object named Users, and a child object named User. The child object had an attribute called Name, with a value of Joe. So when I asked PowerShell to display \$xml.users.user, it displays all the attributes for that user. I can prove that PowerShell treated \$xml as XML data by using Get-Member:

```
PSH C:\> $xml | get-member

        TypeName: System.Xml.XmlDocument

Name                MemberType          Definition
-----                -
ToString            CodeMethod          static System.Stri
add_NodeChanged     Method              System.Void add_No
add_NodeChanging    Method              System.Void add_No
add_NodeInserted    Method              System.Void add_No
add_NodeInserting   Method              System.Void add_No
add_NodeRemoved     Method              System.Void add_No
add_NodeRemoving    Method              System.Void add_No
AppendChild         Method              System.Xml.XmlNode
Clone               Method              System.Xml.XmlNode
CloneNode           Method              System.Xml.XmlNode
CreateAttribute      Method              System.Xml.XmlAttr
CreateCDATASection  Method              System.Xml.XmlCDat
CreateComment        Method              System.Xml.XmlComm
CreateDocumentFragment Method              System.Xml.XmlDocu
CreateDocumentType  Method              System.Xml.XmlDocu
CreateElement        Method              System.Xml.XmlElem
CreateEntityReference Method              System.Xml.XmlEnti
CreateNavigator      Method              System.Xml.XPath.X
CreateNode           Method              System.Xml.XmlNode
CreateProcessingInstruction Method              System.Xml.XmlProc
...
```

This demonstrates not only that variables *are* objects but also that PowerShell *does* understand different types of data and provides different capabilities for them.

Curious about what object types are available? In PowerShell's installation folder, you'll find a file named Types.mshxml, which lists them all. Each entry in this XML-formatted file looks something like this:

```
<Type>
  <Name>System.Array</Name>
  <Members>
    <AliasProperty>
      <Name>Count</Name>
      <ReferencedMemberName>Length</ReferencedMemberName>
    </AliasProperty>
  </Members>
</Type>
```

So this type name is System.Array. There are *lots* of types, though; for the *most* part you won't need to worry about them unless you need to specifically make sure a string of characters is (as in the previous example) treated as a number.

---

## Advanced .NET in PowerShell

PowerShell exposes almost all the .NET Framework, and the Framework has an unbelievable amount of functionality built into it. Although you might never *want* to, you can certainly leverage this functionality anytime. For example, a major shortcoming of VBScript was that it featured very minimal user interface capabilities. Not so with PowerShell, which can draw on the entire .NET Framework for user interface features. The Framework includes classes for something called *Windows Forms*, which are the bits of the Framework used to construct graphical Windows applications. Ryan Paul at Arstechnica.com found that the Windows Forms classes—contained in a .NET assembly named System.Windows.Forms—could be utilized from within PowerShell:

```
PSH C:\> [Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")

GAC      Version          Location
---      -
True     v2.0.50727       C:\WINDOWS\assembly\GAC_MSIL\System.Windows.Forms\2.0

PSH C:\> $window = new-object Windows.Forms.Form
PSH C:\> $window.text = "This is a dialog box!"
PSH C:\> $button = new-object Windows.Forms.Button
PSH C:\> $button.text = "Close"
PSH C:\> $window.controls.add($button)
PSH C:\> $window.ShowDialog()
```

This example loads the System.Windows.Forms assembly into PowerShell. It then uses the New-Object cmdlet to create a new object of the Windows.Forms.Form type, and assign a value to the resulting object's Text property. It then creates a new Windows.Forms.Button object, assigns a value to its Text property, then adds the button to the window's Controls collection. Finally, it calls the window's ShowDialog() method. No code was added to the button, but click the red "X" icon to close the window and PowerShell displays "Cancel" as the result of the ShowDialog() function. This might not be something you use everyday, and it does certainly require a fairly thorough knowledge of the .NET Framework, but it's a powerful capability to know about in case you ever need it.

## Okay—When Can I Get it? What Can it Do?

Keep in mind that PowerShell will initially ship with Microsoft Exchange Server v12, not with Windows Vista. However, a Windows-flavored version of PowerShell will ship sometime between Windows Vista's release and Longhorn Server's release, probably in 2006 or 2007. In the meantime, the beta of PowerShell provides plenty of tantalizing hints as to what is in store for Windows administrators. And keep in mind that, according to Microsoft developers, future Windows admin GUI tools will be MMC snap-ins *built on top of PowerShell*, making PowerShell the single, consistent, and complete point for command-line automation and scripting. Exciting, isn't it?

---

Consider the following example: Want an inventory of a computer's IP addresses and MAC addresses for its network adapters?

```
get-wmiobject win32_networkadapterconfiguration | select  
ipaddress, macaddress, description | where { $_.macaddress.length  
-gt 0 }
```

And that's not even a "script," it's just a single command line inside PowerShell. Scripts aren't any more complex than a bunch of command lines, like this one, strung together; PowerShell's powerful scripting language lets you have loops and conditional execution—but all that power comes from just about a half-dozen language statements, so the language is a snap to learn.

## Want to Know More?

This eBooklet has just been a brief introduction to PowerShell, how it works, and where it fits into the world of Windows scripting. If you would like to know more—and why wouldn't you?—drop by my Web site at [www.ScriptingAnswers.com](http://www.ScriptingAnswers.com). Together with fellow scripter Jeffery Hicks, I'm working on a book entitled *Microsoft PowerShell: TFM™* (yes, it's TFM that everyone tells you to Read) that will dive into much greater detail. It'll be available in late 2006 from SAPIEN Press ([www.SAPIEN.com](http://www.SAPIEN.com)). Good luck, and start scripting!

## Download Additional eBooks from Realtime Nexus!

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