

Chapter 1

The Windows PowerShell Rap Sheet

In This Chapter

- ▶ Following the birth and evolution of Windows PowerShell
 - ▶ Installing Windows PowerShell 2
 - ▶ Interacting with the Windows PowerShell command shell
 - ▶ Using the Integrated Scripting Environment (ISE)
-

I'm a really lazy person by nature. I'm not lazy in the sense that I like to sit down and do nothing all day long, but rather I hate doing things over and over again. Whenever I find myself doing something very mundane, the first thing that pops into mind is "there has to be a way to automate this!" Computers are great work horses. They can run day in and day out and never complain. Logically, it makes sense to make your computer work for you rather than the other way around, so in my infinite laziness I'm constantly cooking up ways to make my computer work harder so I can have time to do more important things . . . like write this book for you.

Whether you're completely new to scripting or have done some level of automation in the past using other scripting languages, you'll really love Windows PowerShell. It gives Windows users a true shell that provides the same power over the Windows system that only people in the Unix/Linux community enjoyed previously. Microsoft has spent years and years trying to make Windows easier to use, and in the process of doing so have made some things quite frustrating for power users. (Remember when Microsoft was trying to force you to use wizards only?) Windows PowerShell is, in my mind, Microsoft's way of acknowledging that a significant number of users know what they want and don't want to sit around all day long clicking through dialog boxes to get their jobs done.

Addressing the Need for a Powerful, Windows-Focused Scripting Language

You've always had the standard Windows Shell, also known as the *command shell* or the *DOS prompt* (for those who can't let go of the past), to interact with Windows at the command line. You can automate various aspects of Windows from the command shell using built-in commands, other command line applications, and even string them together into Windows Shell *scripts* (or *batch files* for those still clamoring for the good old DOS days). If you want a bit more power and control, you can use Windows Scripting Host (WSH) and then use VBScript or JScript to automate your tasks. So the obvious question is "why add Windows PowerShell to this mix?" After all, can't you accomplish everything you need to do using these existing methods?

Sure, a good portion of everything you need to do in Windows can be accomplished by writing a Windows Shell or WSH script. I've been doing it for years with no problems, and when I first heard of Windows PowerShell being developed several years ago (when it was still under the codename *Monad*) I had mixed feelings. On one hand, it promised a whole new way of doing things, which was exciting, but on the other hand it just became one more thing I needed to learn. As Windows PowerShell came into maturity, I clearly saw that it really did live up to its promises, and I found myself jumping on the Windows PowerShell bandwagon.

Watching Monad morph into PowerShell

Windows PowerShell was architected by Jeffrey P. Snover back in August 2002, under the codename *Monad*. According to the original *Monad Manifesto*, it was designed as the next-generation platform for administrative automation. It was based loosely on the tried and proven approach for administrative automation in Unix.

In traditional command shells, you achieve a desired action by manipulating generally unstructured text output of a previous command to generate the desired output or effect using another command. In a regular Windows Command Shell, for example, you can use the following command sequence to find out if pinging `www.whitehouse.gov` returns any replies.

```
ping www.whitehouse.gov | find "Reply"
```

In the example, you pass the output of the `ping` command against `www.whitehouse.gov` into the `find` command because you want to filter the

output so only the lines containing the word *Reply* get displayed. Monad tackled the limitations of this traditional method by devising a new approach for building commands by leveraging the .NET framework and its object model. Monad does this by defining an automation model where commands called *Cmdlets* (read as *command-lets*) can pass data to each other as structured objects rather than a loose collection of text.

My intent isn't to give you a history lesson on Windows PowerShell but rather to help you understand why it looks and acts the way it does. As you use Windows PowerShell, you might notice, for example, that the command syntax has a Unix feel to it. This isn't by coincidence but rather due to the language being modeled from powerful Unix shells with the added .NET twist. Don't be intimidated, however — PowerShell is one of the easiest scripting languages to use and is very intuitive.



If you want to read the Monad Manifesto as it originally appeared in 2002, you can view it on the Windows PowerShell team blog (<http://blogs.msdn.com/powershell/archive/2007/03/19/monad-manifesto-the-origin-of-windows-powershell.aspx>).

A little bit on Windows PowerShell 1.0

Windows PowerShell brings together the best parts of interacting with the traditional Windows Shell along with the power of writing WSH scripts. It creates a rich command line-based environment that puts more power into your hands by letting you run new PowerShell commands called *Cmdlets*. These are .NET class-based commands that give you the flexibility of high-level scripting while allowing you to access very low-level Application Programming Interfaces (APIs) through .NET wrappers.

Windows PowerShell 1.0 was the first full-production release of Windows PowerShell, and even though it delivered on many of the key elements needed to use it, it was adopted slowly for a few reasons:

- ✓ It wasn't built into any of the existing Windows operating systems, so administrators who wanted to use it had to make a conscious effort to deploy the PowerShell run-time.
- ✓ Administrators who had already mastered existing scripting languages didn't feel the need to use a new shell to accomplish the same tasks.
- ✓ As a new product, it took a while for enough people to start using it before the Windows PowerShell community became proficient enough to be able to demonstrate the more creative ways to use it.

Eventually Microsoft's own developers started taking advantage of Windows PowerShell 1.0, and it was soon adopted in their mainstream products like Microsoft Exchange 2007 and Systems Center Operations Manager (*SCOM*, formerly known as *MOM*). PowerShell 1.0 was then released with Windows Server 2008 as an installable, out-of-box feature. You and I should be excited about this because it really brings Windows PowerShell into the mainstream and also demonstrates Microsoft's commitment to bringing Windows PowerShell into the forefront of its systems management strategies.

Windows PowerShell 2, the Next Evolution

Despite the slow adoption of Windows PowerShell 1.0, a growing Windows PowerShell community emerged and put it through its paces. The Windows PowerShell developers at Microsoft took a lot of this feedback and criticism to produce what promises to be a much more production-worthy scripting environment — Windows PowerShell 2.

I'm sure enough time has now elapsed since you first heard about Windows PowerShell that it has piqued your curiosity (which is probably one of the reasons why you picked up this book). It's a great time for you to discover this scripting language because many of the limitations people faced while working with Windows PowerShell 1.0 have since been worked out. What you're all left with is a much more usable command shell that offers a host of different ways to do things. Your only real limit is your own creativity.

I know you're already asking the obvious: What's new in Windows PowerShell 2 that makes it so special? Here are some of the major changes and enhancements made to Windows PowerShell:

- ✔ **PowerShell remoting:** Gives you the ability to execute Cmdlets and scripts remotely. See Chapter 15.
- ✔ **Background jobs:** As the name implies, this improvement allows you to run commands in the background while you continue to work on other things. See Chapter 15.
- ✔ **Advanced functions:** Cmdlets used to be written only in C# and VB.NET. Now you can write your command pseudo-Cmdlets using Windows PowerShell itself. See Chapter 14.
- ✔ **Data language:** Gives you the ability to separate your code from the data, making it more portable and easier to share.
- ✔ **Script internationalization:** Helps scripts that have to accommodate multiple languages easier to implement. See Chapter 16.

- ✔ **Script debugging:** Finally, real debugging. You can set breakpoints in your scripts so you can halt execution to find out what's going on at a particular point in the script. See Chapter 17.
- ✔ **Some new operators and automatic variables:** Some new operators to make it easier to split and join strings and automatic variables for accessing user interface language information. See Chapter 5.
- ✔ **Additional new Cmdlets:** Mostly to support the preceding features.
- ✔ **Constrained runspaces:** Gives you the ability to constrain what commands and scripts Windows PowerShell can run within a given runspace.
- ✔ **Runspace pools:** You can think of these as ways to manage command execution by pooling together runspaces.
- ✔ **Integrated Scripting Environment (ISE):** A graphical version of the command shell that adds some cool new features such as multi-tabbed panes for working with multiple scripts at the same time. See Chapter 2.
- ✔ **Out-GridView:** You can output the results of your commands in an interactive table where you can then sort, search, and group the results. See Chapter 25.
- ✔ **New PowerShell APIs:** If you're a programmer, you can get to the new features provided in PowerShell directly using these APIs.
- ✔ **Some minor enhancements to existing commands and shell behavior:** Some additional parameters to existing commands have been added to increase functionality.

Even if you haven't used Windows PowerShell in the past, you can tell just by this list of new features that there are some significant enhancements to Windows PowerShell that go beyond the surface. I think Windows PowerShell 2 is a more complete product that still makes it easy for new users like you to master it while leaving plenty of room for you to grow.

What's really amazing is that while I'd classify many of the changes in Windows PowerShell 2 under an advanced feature category, discovering how to use them is a quick and easy thing even for a beginner. Before you know it, and with the help of this really cool book you're reading, you too will be taking advantage of these new features.

Installing Windows PowerShell 2

Words are just words. I know your heart is pumping already and you're about to scream at the top of your lungs "I want to use Windows PowerShell already, stop talking and tell me how!" Because Windows PowerShell 2

doesn't ship with any of the Windows operating systems except Windows 7, you'll generally need to install it first. Luckily, this task is relatively pain-free, so stick with me for a few seconds.



Windows PowerShell 2 is a replacement for Windows PowerShell 1.0. They can't co-exist on the same system, so if you already have Windows PowerShell 1.0 installed, make sure you uninstall it first. **Note:** To uninstall Windows PowerShell 1.0, you might have to select the Show Updates option in the Add/Remove Programs control panel applet for it to be visible.



Windows PowerShell 2 can be installed on both the x86 and x64 platforms of Windows XP with SP3, Windows Server 2003 with SP2, Windows Vista with SP1, Windows Server 2008, and Windows 7.

You install Windows PowerShell 2 using these four simple steps:

1. **Download and install Microsoft .NET Framework 2.0**
2. **Download and install Microsoft .NET Framework 3.5.1.**

Required for Windows PowerShell Integrated Scripting Environment (ISE) and Out-GridView.

3. **Download and install WinRM 2.0 CTP3.**

This is required if you want to take advantage of the remoting and background jobs features.

4. **Download and install Windows PowerShell 2.**

I'm not going to give you step-by-step instructions here because it's a straightforward "next, next, next" installation.

Firing up the Windows PowerShell Command Shell

Congratulations! Now that you've got Windows PowerShell 2 installed, you can finally have some fun.



First, going forward, you might see me referring to Windows PowerShell 2 simply as PSH. Not only will this save me from carpal tunnel syndrome, but Windows PowerShell is often referred to as PSH within Windows PowerShell community, so don't be surprised if you see that abbreviation. (It's also sometimes just called PS.)

Fire up the PSH command shell by choosing Start⇨All Programs⇨Windows PowerShell V2⇨Windows PowerShell V2.



If you're running Windows Vista, you may need to right-click the shortcut and choose the option to run as Administrator (running elevated) even if you have administrative rights on the system if you get access denied errors.

Windows PowerShell 2 launches and the command shell opens, as shown in Figure 1-1. It looks a lot like your old Windows command shell, except that by default the background is blue and the prompt is prefixed by PS. You can run some familiar DOS commands (such as `DIR` and `CD`), and they'll still work, but the output might look a bit different. Also, running some existing command line applications like `XCOPY . EXE` works too! I get into how this all works in future chapters, but the ability to run non-PowerShell commands is one of the greatest things about PSH — you can start using PSH today as a replacement command shell and run your old commands while getting familiar with the new PSH way.



PSH runs your regular command line applications as normal, but the built-in commands such as `CD` and `DIR` are actually aliases to new PSH Cmdlets. This is why the output of `DIR` looks a bit different. Also notice that you can't use the old switches (such as `DIR /W`) with `DIR`. The reason is because the underlying Cmdlet that `DIR` is mapped to uses different parameters. I talk more about aliases in Chapter 2.

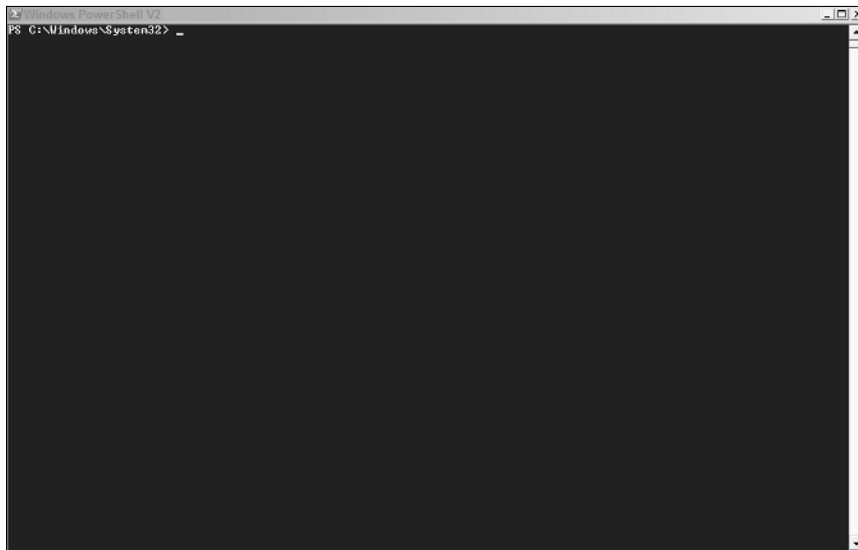


Figure 1-1:
The
Windows
PowerShell
command
shell.

Going GUI: The Windows PowerShell Integrated Shell Environment (ISE)

The Windows PowerShell Integrated Shell Environment (ISE) is a bit of a mouthful, but it's really just a more graphically rich interface (see Figure 1-2) for interacting with PSH. You launch it the same way as the regular PSH command shell (see the preceding section), but you select Windows PowerShell ISE instead; select Start→All Programs→Windows PowerShell v2→Windows PowerShell ISE.



Figure 1-2:
The
Windows
PowerShell
ISE.

Here's what you get with this handsome interface:

- ✓ **Script/Editor pane:** This is where you can view and edit your PSH scripts.
- ✓ **Output pane:** This is where the output of all your command or script is displayed.
- ✓ **Command pane:** You can enter commands in this pane just as you would in a regular PSH command shell.

You can also create PSH scripts by choosing File⇨New to display the editor pane above the output pane. If you're working on multiple scripts, a tabbed interface is displayed so you can easily switch back and forth between the different script windows, as shown in Figure 1-3.

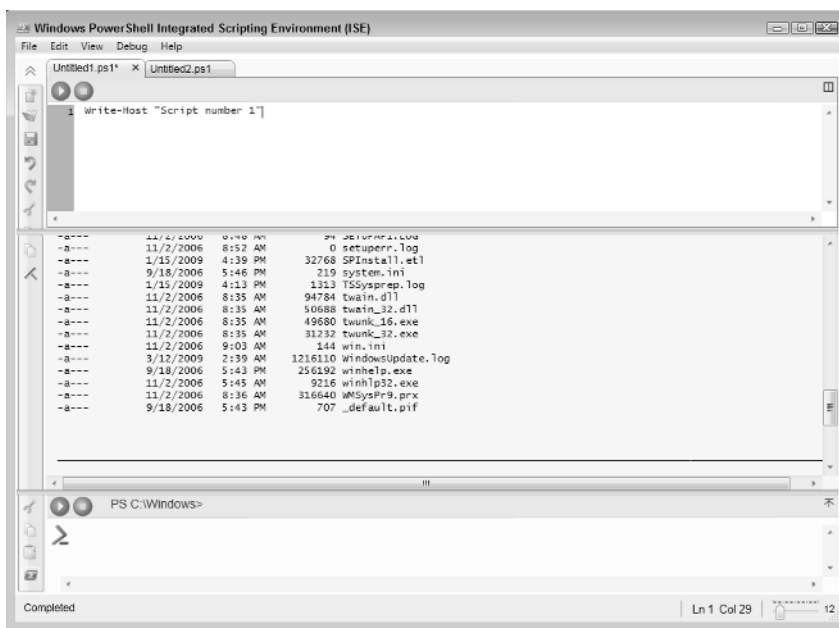


Figure 1-3:
The
Windows
PowerShell
ISE window
with the
tabbed
script editor
interface.

You'll also notice that when you have a script open, you can run it simply by clicking the Run button (the right-pointing triangle, similar to the Play button on a CD player) on the toolbar. The toolbar has all the standard text-editing features as well as syntax highlighting, which makes editing your scripts a bit easier on the eyes. The best part is that the debugger is easily accessible from the Debug menu. (I cover debugging concepts in-depth in Chapter 17.) The ISE is an excellent tool for writing, running, and debugging your scripts in one easy-to-use environment. Think of it as a miniature Visual Studio for Windows PowerShell. I talk more about the ISE in the next chapter.



Although the ISE script pane is primarily designed for writing and editing scripts, it's a pure text editor, so you can use it to open or create plain text files and XML files.