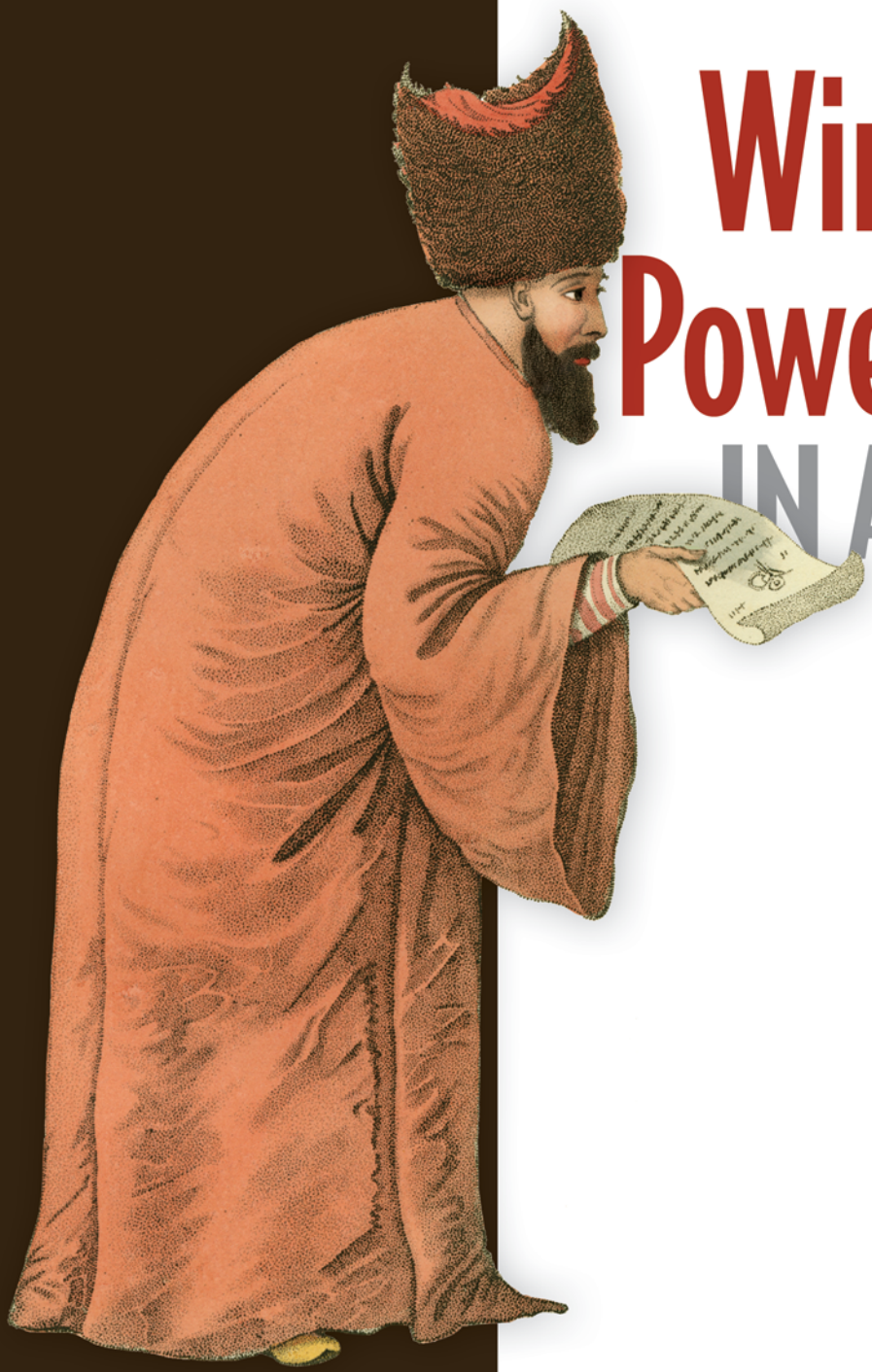
SAMPLE CHAPTER

THIRD EDITION



# Windows PowerShell IN ACTION

Bruce Payette
Richard Siddaway

MANNING

*Windows PowerShell in Action*
*Third Edition*

by Bruce Payette
Richard Siddaway

**Chapter 1**

# *brief contents*

# 1 Welcome to PowerShell

**This chapter covers**

- Core concepts
- Aliases and elastic systems
- Parsing and PowerShell
- Pipelines
- Formatting and output

*Vizzini: Inconceivable!*

*Inigo: You keep on using that word. I do not think it means what you think it means.*

—*William Goldman,* The Princess Bride

It may seem strange to start by welcoming you to PowerShell when PowerShell is ten years old (at the time of writing), is on its fifth version, and you're reading the third edition of this book.

> **NOTE** PowerShell v6 is under development as we write this. The appendix covers the changes that this new version will introduce.

In reality the adoption of PowerShell is only now achieving significant momentum, meaning that to many users PowerShell is a new technology and the three versions

of PowerShell subsequent to this book's second edition contain many new features. Welcome to PowerShell.

> **NOTE** This book is written using PowerShell v5. It'll be noted in the text where earlier versions are different, or work in a different manner. We'll also document when various features were introduced to PowerShell or significantly modified between versions. We treat v5 and v5.1 together as v5 as the differences are relatively minor.

Windows PowerShell is the command and scripting language from Microsoft built into all versions of Windows since Windows Server 2008. Although PowerShell is new and different (or has new features you haven't yet explored), it's been designed to make use of what you already know, making it easy to learn. It's also designed to allow you to learn a bit at a time.

### Running PowerShell commands

You have two choices for running the examples provided in this book. First is to use the PowerShell console. This provides a command-line interface. It's the tool of choice for interactive work.

The second choice is the PowerShell Integrated Scripting Environment (ISE). The ISE supplies an editing pane plus a combined output and interactive pane. The ISE is the tool of choice when developing scripts, functions, and other advanced functionality.

The examples in the book will be written in a way that allows pasting directly into either tool.

Third-party tools exist, such as those supplied by Sapien, but we'll only consider the native tools in this book.

Starting at the beginning, here's the traditional "Hello world" program in PowerShell:

```
'Hello world.'
```

But "Hello world" itself isn't interesting. Here's something a bit more complicated:

```
Get-ChildItem -Path $env:windir\*.log |
Select-String -List error |
Format-Table Path,LineNumber –AutoSize
```

Although this is more complex, you can probably still figure out what it does. It searches all the log files in the Windows directory, looking for the string "error", and then prints the full name of the matching file and the matching line number. "Useful, but not special," you might think, because you can easily do this using

`cmd.exe` on Windows or `bash` on UNIX. What about the "big, really big" thing? Well, how about this example:

```
([xml] [System.Net.WebClient]::new().
    DownloadString('http://blogs.msdn.com/powershell/rss.aspx')).
        RSS.Channel.Item |
            Format-Table title,link
```

Now we're getting somewhere. This script downloads the RSS feed from the PowerShell team blog and then displays the title and a link for each blog entry. By the way, you weren't expected to figure out this example yet. If you did, you can move to the head of the class!

One last example:

```
using assembly  System.Windows.Forms
using namespace System.Windows.Forms
$form = [Form] @{
    Text = 'My First Form'
}
$button = [Button] @{
    Text = 'Push Me!'
    Dock = 'Fill'
}
$button.add_Click{
    $form.Close()
}
$form.Controls.Add($button)
$form.ShowDialog()
```

This script uses the Windows Forms library (WinForms) to build a GUI that has a single button displaying the text "Push Me!" Figure 1.1 shows the window this script creates.

When you click the button, it closes the form and exits the script. With this you go from `"Hello world"` to a GUI application in less than two pages.

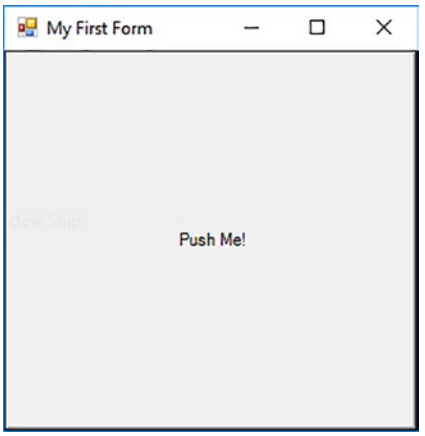



**Figure 1.1 When you run the code from the example, this window will be displayed.**

Let's come back down to Earth for a minute. The intent of chapter 1 is to set the stage for understanding PowerShell—what it is, what it isn't, and, almost as important, why the PowerShell team made the decisions they made in designing the PowerShell language. Chapter 1 covers the goals of the project, along with some of the major issues the team faced in trying to achieve those goals. First, a philosophical digression: while under development, from 2002 until the first public release in 2006, the codename for this project was Monad. The

name Monad comes from *The Monadology* by Gottfried Wilhelm Leibniz, one of the inventors of calculus. Here's how Leibniz defined the Monad:

> *The Monad, of which we shall here speak, is nothing but a simple substance, which enters into compounds. By "simple" is meant "without parts."*
>
> —*Gottfried Wilhelm Leibniz,* The Monadology (translated by Robert Latta)

In *The Monadology*, Leibniz describes a world of irreducible components from which all things could be composed. This captures the spirit of the project: to create a toolkit of simple pieces you compose to create complex solutions.

## 1.1 What is PowerShell?

What is PowerShell, and what can you do with it? Ask a group of PowerShell users and you'll get different answers:

- PowerShell is a command-line shell.
- PowerShell is a scripting environment.
- PowerShell is an automation engine.

These are all part of the answer. We prefer to say PowerShell is a tool you can use to manage your Microsoft-based machines and applications that programs consistency into your management process. The tool is attractive to administrators and developers in that it can span the range of command line, simple and advanced scripts, to real programs.

> **NOTE** If you take this to mean PowerShell is the ideal DevOps tool for the Microsoft platform, then congratulations—you've got it in one.

PowerShell draws heavily from existing command-line shell and scripting languages, but the language, runtime, and subsequent additions, such as PowerShell Workflows and Desired State Configuration, were designed from scratch to be an optimal environment for the modern Windows operating system.

Most people are introduced to PowerShell through its interactive aspects. Let's refine our definitions of shell and scripting.

### 1.1.1 Shells, command lines, and scripting languages

In the previous section we called PowerShell a command-line shell. You may be asking, what's a shell? And how's it different from a command interpreter? What about scripting languages? If you can script in a shell language, doesn't that make it a scripting language? In answering these questions, let's start with shells.

Defining a shell can be tricky because pretty much everything at Microsoft has something called a *shell*. Windows Explorer is a shell. Visual Studio has a component called a shell. Heck, even the Xbox has something called a shell.

Historically, the term *shell* describes the piece of software that sits over an operating system's core functionality. This core functionality is known as the *operating system kernel* (shell ... kernel ... get it?). A shell is the piece of software that lets you access the functionality provided by the operating system. For our purposes, we're more interested in the traditional text-based environment where the user types a command and receives a response. Put another way, a shell is a command-line interpreter. The two terms can be used for the most part interchangeably.

### Scripting languages vs. shells

If this is the case, what's scripting and why are scripting languages not shells? To some extent, there's no difference. Many scripting languages have a mode in which they take commands from the user and then execute those commands to return results. This mode of operation is called a *read-evaluate-print loop*, or REPL. In what way is a scripting language with a REPL not a shell? The difference is mainly in the user experience. A proper command-line shell is also a proper UI. As such, a command line has to provide a number of features to make the user's experience pleasant and customizable, including aliases (shortcuts for hard-to-type commands), wildcard matching to avoid having to type out full names, and the ability to start other programs easily. Finally, command-line shells provide mechanisms for examining, editing, and re-executing previously typed commands. These mechanisms are called *command history*.

If scripting languages can be shells, can shells be scripting languages? The answer is, emphatically, yes. With each generation, the UNIX shell languages have grown increasingly powerful. It's possible to write substantial applications in a modern shell language, such as Bash or Zsh. Scripting languages characteristically have an advantage over shell languages in that they provide mechanisms to help you develop larger scripts by letting you break a script into components, or *modules*. Scripting languages typically provide more sophisticated features for debugging your scripts. Next, scripting language runtimes are implemented in a way that makes their code execution more efficient, and scripts written in these languages execute more quickly than they would in the corresponding shell script runtime. Finally, scripting language syntax is oriented more toward writing an application than toward interactively issuing commands.

In the end, there's no hard-and-fast distinction between a shell language and a scripting language. Because PowerShell's goal is to be both a good scripting language and a good interactive shell, balancing the trade-offs between user experience and script authoring was one of the major language design challenges.

### Managing Windows through objects

Another factor that drove the need for a new shell model is, as Windows acquired more and more subsystems and features, the number of issues users had to think about when managing a system increased dramatically. To help users deal with this increase in complexity, the manageable elements were factored into structured data objects. This collection of *management objects* is known internally at Microsoft as the *Windows Management Surface*.

> **NOTE** Microsoft wasn't the only company running into issues caused by increased complexity. Most people in the industry were having this problem. This led to the Distributed Management Task Force (dmtf.org), an industry organization, creating a standard for management objects called the *Common Information Model* (CIM). Microsoft's original implementation of this standard is called *Windows Management Instrumentation* (WMI).

Although this factoring addressed overall complexity and worked well for GUIs, it made it much harder to work with using a traditional text-based shell environment.

Windows is an API-driven operating system, compared to UNIX and its derivatives, which are document (or text) driven. You can administer UNIX by changing configuration files. In Windows, you need to use the API, which means accessing properties and using methods on the appropriate object.

Finally, as the power of the PC increased, Windows began to move off the desktop and into the corporate datacenter. In the corporate datacenter, there were a large number of servers to manage, and the graphical point-and-click management approach didn't scale. All these elements combined to make it clear Microsoft could no longer ignore the command line.

Now that you grasp the environmental forces that led to the creation of PowerShell—the need for command-line automation in a distributed object-based operating environment—let's look at the form the solution took.

## 1.2 PowerShell example code

We've said PowerShell is for solving problems that involve writing code. By now you're probably asking "Dude! Where's my code?" Enough talk, let's see some example code! First, we'll revisit the `Get-ChildItem` example. This time, instead of displaying the directory listing, you'll save it into a file using output redirection like in other shell environments. In the following example, you'll use `Get-ChildItem` to get information about a file named somefile.txt in the root of the C: drive. Using redirection, you'll direct the output into a new file, c:\foo.txt, and then use the `type` command to display what was saved. Here's what this looks like:

```
PS> Get-ChildItem -Path C:\somefile.txt

    Directory: C:\

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        29/05/2017     13:58          25424 somefile.txt
```

> **NOTE** PowerShell has aliases for many cmdlets so `dir C:\somefile.txt` and `ls C:\somefile.txt` would both work. It is best practice to reserve aliases for interactive usage and not use them in scripts. We'll usually use the full cmdlet name but may occasionally use aliases to save space.

Next, instead of displaying the directory listing, you'll save it into a file using output redirection as in other shell environments. In the following example, you'll get

information about a file named somefile.txt in the root of the C: drive. Using redirection, you direct the output into a new file, c:\foo.txt, and then use the `Get-Content` (you can use the alias of `cat` or `type` if you prefer) command to display what was saved. Here's what this looks like:

```
PS> Get-ChildItem -Path C:\somefile.txt > c:\foo.txt
PS> Get-Content -Path C:\foo.txt

    Directory: C:\

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----       29/05/2017     13:58          25424 somefile.txt
```

As you can see, commands work more or less as you'd expect. Let's go over other things that should be familiar to you.

> **NOTE** On your system choose any file that exists and the example will work fine, though obviously, the output will be different.

### 1.2.1 Navigation and basic operations

The PowerShell commands for working with the file system should be pretty familiar to most users. You navigate around the file system with the `cd` (alias for `Set-Location`) command. Files are copied with the `copy` or `cp` (aliases for `Copy-Item`) commands, moved with the `move` and `mv` (aliases for `Move-Item`) commands, and removed with the `del` or `rm` (aliases for `Remove-Item`) commands. Why two of each command? One set of names is familiar to `cmd.exe`/DOS users and the other is familiar to UNIX users. In practice, they're aliases for the same command, designed to make it easy for people to get going with PowerShell.

> **NOTE** In PowerShell v6 Core on Linux or macOS these common aliases have been removed to prevent conflict with native commands on Linux and macOS. The aliases are present in the Windows versions of PowerShell v6 Core.

Keep in mind that, although the commands are similar, they're not exactly the same as either of the other two systems. You can use the `Get-Help` command to get help about these commands. Here's the output of `Get-Help` for the `dir` command:

```
PS> Get-Help dir

NAME
    Get-ChildItem

SYNOPSIS
    Gets the items and child items in one or more specified locations.

SYNTAX
    Get-ChildItem [[-Filter] <String>] [-Attributes {ReadOnly |
 Hidden | System | Directory | Archive | Device |  Normal |
 Temporary | SparseFile | ReparsePoint | Compressed | Offline |
  NotContentIndexed | Encrypted |IntegrityStream | NoScrubData}]
```

```
[-Depth <UInt32>] [-Directory] [-Exclude <String[]>] [-File]
[-Force] [-Hidden][-Include <String[]>] -LiteralPath <String[]>
[-Name] [-ReadOnly] [-Recurse] [-System] [-UseTransaction]
   [<CommonParameters>]

    Get-ChildItem [[-Path] <String[]>] [[-Filter] <String>]
[-Attributes {ReadOnly | Hidden | System | Directory |
Archive | Device | Normal | Temporary | SparseFile |
ReparsePoint | Compressed | Offline | NotContentIndexed |
Encrypted | IntegrityStream | NoScrubData}] [-Depth <UInt32>]
[-Directory] [-Exclude <String[]>] [-File] [-Force]
[-Hidden] [-Include <String[]>] [-Name] [-ReadOnly] [-Recurse]
[-System] [-UseTransaction] [<CommonParameters>]

DESCRIPTION
    The Get-ChildItem cmdlet gets the items in one or more specified
   locations. If the item is a container, it gets the items inside the
   container, known as child items. You can use the Recurse parameter to get
   items in all child containers.

    A location can be a file system location, such as a directory, or a
   location exposed by a different Windows PowerShell provider, such as a
   registry hive or a certificate store.

RELATED LINKS
    Online Version: http://go.microsoft.com/fwlink/?LinkId=821580
    Get-Item
    Get-Location
    Get-Process
    Get-PSProvider

REMARKS
    To see the examples, type: "get-help Get-ChildItem -examples".
    For more information, type: "get-help Get-ChildItem -detailed".
    For technical information, type: "get-help Get-ChildItem -full".
For online help, type "get-help Get-ChildItem -online"PowerShell help system
```

### PowerShell help system

The PowerShell help subsystem contains information about all the commands provided with the system and is a great way to explore what's available.

In PowerShell v3 and later, help files aren't installed by default. Help has become updatable and you need to install the latest versions yourself. See `Get-Help about_Updatable_Help`.

You can even use wildcard characters to search through the help topics (v2 and later). This is the simple text output. The PowerShell ISE also includes help in the richer Windows format and will let you choose an item and then press F1 to view the help for the item. By using the `-Online` option to `Get-Help`, you can view the help text for a command or topic using a web browser.

```
PS> Get-Help Get-ChildItem
```

***(Continued)***

displays the information in the help file stored locally.

```
PS> Get-Help Get-ChildItem -Online
```

displays the online version of the help file.

Using the `-Online` option is the best way to get help because the online documentation is constantly being updated and corrected, whereas the local copies aren't.

### 1.2.2 Basic expressions and variables

In addition to running commands, PowerShell can evaluate expressions. In effect, it operates as a kind of calculator. Let's evaluate a simple expression:

```
PS> 2+2
4
```

Notice as soon as you typed the expression, the result was calculated and displayed. It wasn't necessary to use any kind of print statement to display the result. It's important to remember whenever an expression is evaluated, the result of the expression is output, not discarded. PowerShell supports most of the basic arithmetic operations you'd expect, including floating point.

You can save the output of an expression to a file by using the redirection operator:

```
PS> (2+2)*3/7 > c:\foo.txt
PS> Get-Content c:\foo.txt
1.71428571428571
```

Saving expressions into files is useful; saving them in variables is more useful:

```
PS> $n = (2+2)*3
PS> $n
12

PS> $n / 7
1.71428571428571
```

Variables can also be used to store the output of commands:

```
PS> $files = Get-ChildItem
PS> $files[1]

    Directory: C:\Users\Richard\Documents

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         16/02/2017     18:36            Custom Office Templates
```

In this example, you extracted the second element of the collection of file information objects returned by the `Get-ChildItem` command. You were able to do this because you saved the output of the `Get-ChildItem` command as an array of objects in the `$files` variable.

> **NOTE** Collections in PowerShell start at 0, not 1. This is a characteristic we've inherited from .NET. This is why `$files[1]` extracts the second element, not the first.

Given PowerShell is all about objects, the basic operators need to work on more than numbers. Chapters 3 and 4 cover these features in detail.

### 1.2.3 Processing data

As you've seen, you can run commands to get information, perform some basic operations on this information using the PowerShell operators, and then store the results in files and variables. Let's look at additional ways you can process this data. First, you'll see how to sort objects and how to extract properties from those objects. Then we'll look at using the PowerShell flow-control statements to write scripts that use conditionals and loops to do more sophisticated processing.

#### SORTING OBJECTS

First, sort the list of file information objects returned by `Get-ChildItem`. Because you're sorting objects, the command you'll use is `Sort-Object`. For convenience, you'll use the shorter alias `sort` in these examples. Start by looking at the default output, which shows the files sorted by filename:

```
PS> cd c:\files
PS> Get-ChildItem

    Directory: C:\files

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---        21/01/2015     18:10          9 File 1.txt
-a---        11/07/2015     15:14      15986 File 2.txt
-a---        21/01/2015     18:10          9 File 3.txt
-a---        21/01/2015     18:10          9 File 4.txt
```

The output shows the basic properties on the file system objects, sorted by filename. Now sort by filename in descending order:

```
PS> Get-ChildItem | sort -Descending

            Directory: C:\files

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---        21/01/2015     18:10          9 File 4.txt
-a---        21/01/2015     18:10          9 File 3.txt
-a---        11/07/2015     15:14      15986 File 2.txt
-a---        21/01/2015     18:10          9 File 1.txt
```

There you have it—files sorted by filename in reverse order. Now you'll sort by something other than the filename: file length.

> **NOTE** Many examples in this book use aliases (shortcuts) rather than the full cmdlet name. This is for brevity and to ensure the code fits neatly in the page.

In PowerShell, when you use the `Sort-Object` cmdlet (alias `sort`), you don't have to tell it to sort numerically—it already knows the type of the field, and you can specify the sort key by property name instead of a numeric field offset. The result looks like this:

```
PS> Get-ChildItem | sort -Property length

        Directory: C:\files

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         21/01/2015     18:10          9 File 3.txt
-a---         21/01/2015     18:10          9 File 4.txt
-a---         21/01/2015     18:10          9 File 1.txt
-a---         11/07/2015     15:14      15986 File 2.txt
```

This illustrates what working with pipelines of objects gives you:

- You have the ability to access data elements by name instead of using substring indexes or field numbers.
- By having the original type of the element preserved, operations execute correctly without you having to provide additional information.

Now let's look at other things you can do with objects.

#### SELECTING PROPERTIES FROM AN OBJECT

In this section we'll introduce another cmdlet for working with objects: `Select-Object`. This cmdlet allows you to select a subrange of the objects piped into it and specify a subset of the properties on those objects.

Say you want to get the largest file in a directory and put it into a variable:

```
PS> $a = Get-ChildItem | sort -Property length -Descending |
Select-Object -First 1
PS> $a

        Directory: C:\files

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         11/07/2015     15:14      15986 File 2.txt
```

> **NOTE** You'll notice the secondary prompt `>>` when you copy the previous example into a PowerShell console. The first line of the command ended in a pipe symbol. The PowerShell interpreter noticed this, saw the command was incomplete, and prompted for additional text to complete the command. Once the command is complete, you type a second blank line to send the command to the interpreter. If you want to cancel the command, you can

press Ctrl-C at any time to return to the normal prompt. The code examples in the book won't include the >> to make copying from the electronic version simpler for the reader.

Now say you want only the name of the directory containing the file and not all the other properties of the object. You can also do this with `Select-Object` (alias `select`). As with the `Sort-Object` cmdlet, `Select-Object` takes a `-Property` parameter (you'll see this frequently in the PowerShell environment—commands are consistent in their use of parameters):

```
PS> $a = Get-ChildItem| sort -Property length -Descending |
Select-Object -First 1 -Property Directory
PS> $a

Directory
---------
C:\files
```

You now have an object with a single property.

#### PROCESSING WITH THE FOREACH-OBJECT CMDLET

The final simplification is to get the value itself. We'll introduce a new cmdlet that lets you do arbitrary processing on each object in a pipeline. The `ForEach-Object` cmdlet executes a block of statements for each object in the pipeline. You can get an arbitrary property out of an object and then do arbitrary processing on that information using the `ForEach-Object` command. Here's an example that adds up the lengths of all the objects in a directory:

```
PS> $total = 0
PS> Get-ChildItem | ForEach-Object {$total += $_.length }
PS> $total
16013
```

In this example you initialize the variable `$total` to `0`, then add to it the length of each file returned by the `Get-ChildItem` command, and display the total (you'll get a different total on your system).

#### PROCESSING OTHER KINDS OF DATA

One of the great strengths of the PowerShell approach is once you learn a pattern for solving a problem, you can use this same pattern over and over again. Say you want to find the largest three files in a directory. The command line might look like this:

```
PS> Get-ChildItem | sort -Descending length | select -First 3
```

Here, the `Get-ChildItem` command retrieved the list of file information objects, PowerShell then sorted them in descending order by length, and then selected the first three results to get the three largest files.

Now let's tackle a different problem. You want to find the three processes on the system with the largest working set size. Here's what this command line looks like:

```
PS> Get-Process | sort -Descending ws | select -First 3
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1337    1916   235360     287852  1048    63.23   2440 WWAHost
    962      55    94460     176008   692   340.25   6632 WINWORD
    635      40   136040     140088   783     6.42   2564 powershell
```

This time you run `Get-Process` to get data about the processes on this computer, and sort on the working set instead of the file size. Otherwise, the pattern is identical to the previous example. This command pattern can be applied over and over.

> **NOTE** Because of the ability to apply a command pattern repeatedly, most of the examples in this book are deliberately generic. The intent is to highlight the pattern of the solution rather than show a specific example. Once you understand the basic patterns, you can effectively adapt them to solve a multitude of other problems.

### 1.2.4 Flow-control statements

Pipelines are great, but sometimes you need more control over the flow of your script. PowerShell has the usual flow-control statements found in most programming languages. These include the basic `if` statements, a powerful `switch` statement, and loops like `while`, `for` and `foreach`, and so on. Here's an example showing the `while` and `if` statements:

```
PS> $i=0
PS> while ($i++ -lt 10) { if ($i % 2) {"$i is odd"}}
1 is odd
3 is odd
5 is odd
7 is odd
9 is odd
```

This example uses the `while` loop to count through a range of numbers, printing only the odd numbers. In the body of the `while` loop is an `if` statement that tests to see whether the current number is odd, and then writes a message if it is. You can do the same thing using the `foreach` statement and the range operator (`..`), but much more succinctly:

```
PS> foreach ($i in 1..10) { if ($i % 2) {"$i is odd"}}
```

The `foreach` statement iterates over a collection of objects, and the range operator is a way to generate a sequence of numbers. The two combine to make looping over a sequence of numbers a very clean operation.

Because the range operator generates a sequence of numbers, and numbers are objects like everything else in PowerShell, you can implement this using pipelines and the `ForEach-Object` (alias `foreach`) cmdlet:

```
PS> 1..10 | foreach { if ($_ % 2) {"$_ is odd"}}
```

These examples only scratch the surface of what you can do with the PowerShell flow-control statements. (Wait until you see the `switch` statement!) The complete set of control structures is covered in detail in chapter 5 with lots of examples.

### 1.2.5 Scripts and functions

What good is a scripting language if you can't package commands into scripts? PowerShell lets you do this by putting your commands into a text file with a .ps1 extension and then running that command. You can even have parameters in your scripts. Put the following text into a file called hello.ps1:

```
param($name = 'bub')
"Hello $name, how are you?"
```

Notice the `param` keyword is used to define a parameter called `$name`. The parameter is given a default value of `'bub'`. Now you can run this script from the PowerShell prompt by typing the name as `.\hello`. You need the `.\` to tell PowerShell to get the command from the current directory.

> **NOTE** Before you can run scripts on a machine in the default configuration, you'll have to change the PowerShell execution policy to allow scripts to run. Use `Get-Help about_execution_policies` to view detailed instructions on execution policies. The default settings change between Windows versions, so be careful to check the execution policy setting.

The first time you run this script, you won't specify any arguments:

```
PS> .\hello
Hello bub, how are you?
```

You see the default value was used in the response. Run it again, but this time specify an argument:

```
PS> .\hello Bruce
Hello Bruce, how are you?
```

Now the argument is in the output instead of the default value. Sometimes you want to have subroutines in your code. PowerShell addresses this need through functions. Let's turn the hello script into a function. Here's what it looks like:

```
function hello {
param($name = "bub")
"Hello $name, how are you"
}
```

The body of the function is exactly the same as the script. The only thing added is the `function` keyword, the name of the function, and braces around the body of the function. Now run it, first with no arguments as you did with the script

```
PS> hello
Hello bub, how are you
```

and then with an argument:

```
PS> hello Bruce
Hello Bruce, how are you
```

Obviously, the function operates in the same way as the script, except PowerShell didn't have to load it from a disk file, making it a bit faster to call. Scripts and functions are covered in detail in chapter 6.

### 1.2.6 Remote administration

In the previous sections, you've seen the kinds of things you can do with PowerShell on a single computer, but the computing industry has long since moved beyond a one-computer world. Being able to manage groups of computers, without having to physically visit each one, is critical in the modern cloud-orientated IT world where your server may easily be on another continent. To address this, PowerShell has built-in remote execution capabilities (remoting) and an execution model that ensures if a command works locally it should also work remotely.

> **NOTE** Remoting was introduced in PowerShell v2. It isn't available in PowerShell v1.

The core of PowerShell remoting is `Invoke-Command` (aliased to `icm`). This command allows you to invoke a block of PowerShell script on the current computer, on a remote computer, or on a thousand remote computers. Let's see some of this in action. Microsoft releases patches for Windows on a regular basis. Some of those patches are critical, in that they resolve security-related issues, and as an administrator you need to be able to test if the patch has been applied to the machines for which you're responsible. Checking a single machine is relatively easy—you can use the Windows update option in the control panel and view the installed updates as shown in figure 1.2.

Alternatively, you can use the `Get-HotFix` cmdlet:

```
PS> Get-HotFix -Id KB3213986

Source  Description     HotFixID  InstalledBy         InstalledOn
------  -----------     --------  -----------         -----------

W510W16 Security Update KB3213986 NT AUTHORITY\SYSTEM 12/01/2017 00:00:00
```

This shows you the hotfix is installed on the local machine.

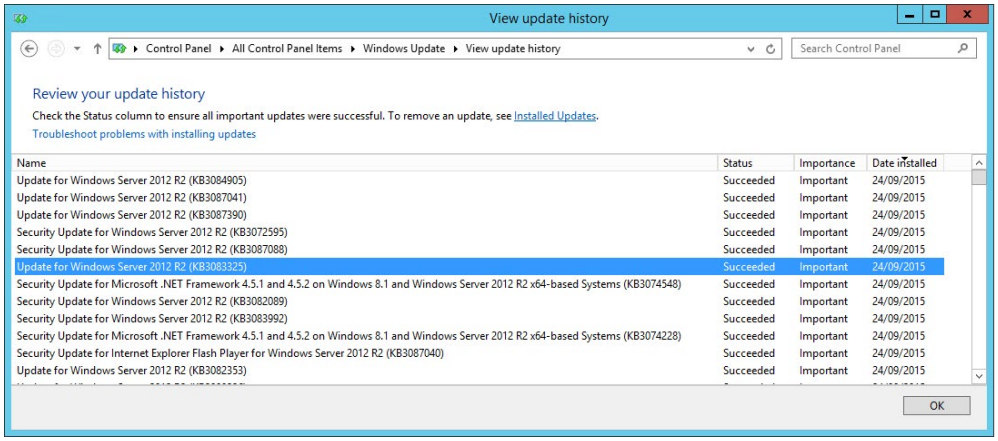



**Figure 1.2 Viewing the installed updates on the local (Windows Server 2012 R2) machine**

> **NOTE** Updates for Windows 10 and Windows Server 2016 tend to be cumulative so your machine may not have KB3213986 installed.

But what about all your other machines? Connecting to each one individually and using the control panel or running the `Get-HotFix` cmdlet is tedious. You need a method of running the cmdlet on remote machines and having the results returned to your local machine.

`Invoke-Command` is used to wrap the previous command:

```
PS> Invoke-Command -ScriptBlock {Get-HotFix -Id KB3213986} `
-ComputerName W16DSC01

Description          : Security Update
HotFixID             : KB3213986
InstalledBy          : NT AUTHORITY\SYSTEM
InstalledOn          : 11/01/2017 00:00:00
PSComputerName       : W16DSC01
```

> **NOTE** `Get-HotFix` has a `-ComputerName` parameter, and, like many cmdlets, is capable of working directly with remote machines. Cmdlet-based remoting often uses protocols other than WS-MAN. Using `Invoke-Command`, as in a PowerShell remoting session, is more efficient, as you'll see in chapter 11.

You have many machines that need testing. Typing in the computer names one at a time is still too tedious. You can create a list of computers, either from a text file or in your code, and test them all:

```
PS>  $computers = 'W16DSC01', 'W16DSC02'
PS>  Invoke-Command -ScriptBlock {Get-HotFix -Id KB3213986} `
-ComputerName $computers |
Format-Table HotFixId, InstalledOn, PSComputerName -AutoSize
```

```
HotFixID  InstalledOn         PSComputerName
--------  -----------         --------------
KB3213986 11/01/2017 00:00:00 W16DSC02
KB3213986 11/01/2017 00:00:00 W16DSC01
```

An error is generated on a computer that doesn't have the patch installed, and results appear on the computers that do.

> **NOTE** In a production script you'd put error handling in place to catch the error and report that the patch wasn't installed. This will be covered in chapter 14.

`Invoke-Command` is the way to programmatically execute PowerShell commands on a remote machine. When you want to connect to a machine to interact with it on a one-to-one basis, you use the `Enter-PSSession` command. This command allows you to start an interactive one-to-one session with a remote computer. Running `Enter-PSSession` looks like this:

```
PS>  Enter-PSSession -ComputerName W16DSC01
[W16DSC01]: PS C:\Users\Richard\Documents> Get-HotFix -Id KB3213986 |
  Format-Table -AutoSize

Source   Description     HotFixID  InstalledBy         InstalledOn
------   -----------     --------  -----------         -----------
W16DSC01 Security Update KB3213986 NT AUTHORITY\SYSTEM 11/01/2017 00:00:00

[W16DSC01]: PS C:\Users\Richard\Documents> Get-Date

05 March 2017 15:35:07

[W16DSC01]: PS C:\Users\Richard\Documents> Exit-PSSession
PS>
```

When you connect to the remote computer, your prompt changes to indicate you're working remotely. Once connected, you can interact with the remote computer the same way you would a local machine. When you're done, exit the remote session with the `Exit-PSSession` command, which returns you to the local session. This brief introduction covers some powerful techniques, but we've only begun to cover all the things remoting lets you do.

At this point, we'll end our "cook's tour" of PowerShell. We've only breezed over the features and capabilities of the environment. In upcoming chapters, we'll explore each of the elements discussed here in detail and a whole lot more.

## 1.3 Core concepts

The core PowerShell language is based on the mature IEEE standard POSIX 1003.2 grammar for the Korn shell, which has a long history as a successful basis for modern shells like Bash and Zsh. The language design team (Jim Truher and Bruce Payette) deviated from this standard where necessary to address the specific needs of an object-based shell and to make it easier to write sophisticated scripts.

PowerShell syntax is aligned with C#. The major value this brings is PowerShell code can be migrated to C# when necessary for performance improvements, and, more importantly, C# examples can be easily converted to PowerShell—the more examples you have in a language, the better off you are.

### 1.3.1 Command concepts and terminology

Much of the terminology used in PowerShell will be familiar if you've used other shells in the Linux or Windows world. Because PowerShell is a new kind of shell, there are a number of terms that are different and a few new terms to learn. In this section, we'll go over the PowerShell-specific concepts and terminology for command types and command syntax.

### 1.3.2 Commands and cmdlets

Commands are the fundamental part of any shell language; they're what you type to get things done. A simple command looks like this:

```
command -parameter1 -parameter2 argument1 argument2
```

A more detailed illustration of the anatomy of this command is shown in figure 1.3. This figure calls out all the individual elements of the command.

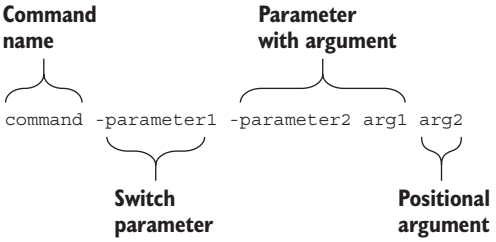



**Figure 1.3 The anatomy of a basic command. It begins with the name of the command, followed by parameters. These may be switch parameters that take no arguments, regular parameters that take arguments, or positional parameters where the matching parameter is inferred by the argument's position on the command line.**

All commands are broken down into the command name, the parameters specified to the command, and the arguments to those parameters. You can think of a parameter as the receiver of a piece of information and the argument as the information itself.

> **NOTE** The distinction between *parameter* and *argument* may seem a bit strange from a programmer's perspective. If you're used to languages such as Python and Visual Basic, which allow for keyword parameters, PowerShell parameters correspond to the keywords, and arguments correspond to the values.

The first element in the command is the name of the command to be executed. The PowerShell interpreter looks at this name and determines which command to run, and which *kind* of command to run. In PowerShell there are a number of categories of commands: cmdlets, shell function commands, script commands, workflow commands, and native Windows commands. Following the command name come zero or

more parameters and/or arguments. A parameter starts with a dash followed by the name of the parameter. An argument, conversely, is the value that will be associated with, or *bound to*, a specific parameter. Let's look at an example:

```
PS> Write-Output -InputObject Hello
Hello
```

Here, the command is `Write-Output`, the parameter is `-InputObject`, and the argument is `Hello`.

What about the positional parameters? When a PowerShell command is created, the author of that command specifies information that allows PowerShell to determine which parameter to bind an argument to, even if the parameter name itself is missing. For example, the `Write-Output` command has been defined such that the first parameter is `-InputObject`. This lets you write:

```
PS> Write-Output Hello
Hello
```

The piece of the PowerShell interpreter that figures all this out is called the *parameter binder*. The parameter binder is smart—it doesn't require you to specify the full name of a parameter as long as you specify enough for it to uniquely distinguish what you mean.

> **NOTE** PowerShell isn't case-sensitive but we use the correct casing on commands and parameters to aid reading. It's also a good practice when scripting, as it's easier to understand the code when you revisit it many months later.

What else does the parameter binder do? It's in charge of determining how to match the types of arguments to the types of parameters. Remember PowerShell is an object-based shell. Everything in PowerShell has a type. PowerShell uses a fairly complex type-conversion system to correctly put things together. When you type a command at the command line, you're typing strings. What happens if the command requires a different type of object? The parameter binder uses the type converter to try to convert that string into the correct type for the parameter. If you use a value that can't be converted to the correct type you get an error message explaining the type conversion failed. We discuss this in more detail in chapter 2 when we talk about types.

What happens if the argument you want to pass to the command starts with a dash? This is where the quotes come in. Let's use `Write-Output` to print out the string "-InputObject":

```
PS> Write-Output -InputObject "-InputObject"
-InputObject
```

And it works as desired. Alternatively, you could type this:

```
PS> Write-Output "-InputObject"
-InputObject
```

The quotes keep the parameter binder from treating the quoted string as a parameter.

Another, less frequently used way of doing this is by using the special "end-of-parameters" parameter, which is two hyphens back to back (`--`). Everything after this sequence will be treated as an argument, even if it looks like a parameter. For example, using `--` you can also write out the string "-InputObject" without using quotes:

```
PS> Write-Output -- -InputObject
-InputObject
```

This is a convention standardized in the POSIX Shell and Utilities specification.

The final element of the basic command pattern is the *switch parameter*. These are parameters that don't require an argument. They're usually either present or absent (obviously they can't be positional). A good example is the `-Recurse` parameter on the `Get-ChildItem` command. This switch tells the `Get-ChildItem` command to display files from a specified directory as well as all its subdirectories:

```
PS> Get-ChildItem -Recurse -Filter c*d.exe C:\Windows

    Directory: C:\Windows\System32

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        11/11/2016     09:56         187520 CloudStorageWizard.exe
-a----        16/07/2016     12:42         232960 cmd.exe
```

As you can see, the `-Recurse` switch takes no arguments. We've only shown the first folder's worth of results for brevity.

> **NOTE** Although it's almost always the case that switch parameters don't take arguments, it's possible to specify arguments to them. We'll save our discussion of when and why you might do this for chapter 7, which focuses on scripts (shell functions and scripts are the only time you need this particular feature, and we'll keep you in suspense for the time being).

Now that we've covered the basic anatomy of the command line, let's go over the types of commands that PowerShell supports.

### 1.3.3 Command categories

As we mentioned earlier, there are four categories of commands in PowerShell: cmdlets, functions, scripts, and native Win32 executables. PowerShell v4, and later, also has configurations (see chapter 18).

#### CMDLETS

The first category of command is a cmdlet (pronounced "command-let"). *Cmdlet* is a term that's specific to the PowerShell environment. A cmdlet is implemented by a .NET class that derives from the `Cmdlet` base class in the PowerShell Software Developers Kit (SDK).

> **NOTE** Building cmdlets is a developer task and requires the PowerShell SDK. This SDK is freely available for download from Microsoft and includes extensive documentation along with many code samples. Our goal is to coach you to effectively use and script in the PowerShell environment, so we're not going to do much more than mention the SDK in this book.

This category of command is compiled into a dynamic link library (DLL) and then loaded into the PowerShell process, usually when the shell starts up. Because the compiled code is loaded into the process, it's the most efficient category of command to execute.

Cmdlets always have names of the form `Verb-Noun`, where the verb specifies the action and the noun specifies the object on which to operate. In traditional shells, cmdlets correspond most closely to what's usually called a *built-in command*. In PowerShell, though, anybody can add a cmdlet to the runtime, and there isn't any special class of built-in commands.

#### FUNCTIONS

The next type of command is a *function*. This is a named piece of PowerShell script code that lives in memory as the interpreter is running, and is discarded on exit. Functions consist of user-defined code that's parsed when defined. This parsed representation is preserved so it doesn't have to be reparsed every time it's used.

Functions in PowerShell v1 could have named parameters like cmdlets but were otherwise fairly limited. In v2 and later, this was fixed, and scripts and functions now have the full parameter specification capabilities of cmdlets. The same basic structure is followed for both types of commands. Functions and cmdlets have the same streaming behavior.

PowerShell workflows were introduced in PowerShell v3. Their syntax is similar to that of a function. When the workflow is first loaded in memory a PowerShell function is created that can be viewed through the function: PowerShell drive. Workflows are covered in chapter 12.

#### SCRIPTS

A *script command* is a piece of PowerShell code that lives in a text file with a .ps1 extension. These script files are loaded and parsed every time they're run, making them somewhat slower than functions to start (although once started, they run at the same speed). In terms of parameter capabilities, shell function commands and script commands are identical.

#### NATIVE COMMANDS (APPLICATIONS)

The last type of command is called a *native command*. These are external programs (typically executables) that can be executed by the operating system. Because running a native command involves creating a whole new process for the command, native commands are the slowest of the command types. Also, native commands do their own parameter processing and don't necessarily match the syntax of the other types of commands.

Native commands cover anything that can be run on a Windows computer, so you get a wide variety of behaviors. One of the biggest issues is when PowerShell waits for a command to finish but it keeps on going. Say you're opening a text document at the command line:

```
PS> .\foo.txt
```

You get the prompt back more or less immediately, and your default text editor will pop up (probably `notepad.exe` because that's the default). The program to launch is determined by the file associations that are defined as part of the Windows environment.

> **NOTE** In PowerShell, unlike in `cmd.exe`, you have to prefix a command with `./` or `.\` if you want to run it out of the current directory. This is part of PowerShell's "Secure by Design" philosophy. This particular security feature was adopted to prevent Trojan horse attacks where the user is lured into a directory and then told to run an innocuous command such as `notepad.exe`. Instead of running the system `notepad.exe`, they end up running a hostile program that the attacker has placed in that directory and named `notepad.exe`.

What if you specify the editor explicitly?

```
PS> notepad foo.txt
```

The same thing happens—the command returns immediately. What if you run the command in the middle of a pipeline?

```
PS> notepad foo.txt | sort-object
<exit notepad>
```

This time PowerShell waits for the command to exit before giving you the prompt. This can be handy when you want to insert something such as a graphical form editor in the middle of a script to do processing. This is also the easiest way to make PowerShell wait for a process to exit (you can also use `Wait-Process`). As you can see, the behavior of native commands depends on the type of native command, as well as where it appears in the pipeline.

A useful thing to remember is the PowerShell interpreter itself is a native command: `powershell.exe`. This means you can call PowerShell from within PowerShell. When you do this, a second PowerShell process is created. In practice, there's nothing unusual about this—that's how all shells work. PowerShell doesn't have to do it often, making it much faster than conventional shell languages.

The ability to run a child PowerShell process is particularly useful if you want to have isolation in portions of your script. A separate process means the child script can't impact the caller's environment. This feature is useful enough that PowerShell has special handling for this case, allowing you to embed the script to run inline. If you want to run a fragment of script in a child process, you can by passing the block of script to the child process delimited by braces. Here's an example:

```
PS> powershell { Get-Process *ss } | Format-Table name, handles

Name  Handles
----  -------
csrss     386
csrss     385
lsass    1778
smss       51
```

Two things should be noted in this example: the script code in the braces can be any PowerShell code, and it will be passed through to the new PowerShell process. The special handling takes care of encoding the script in such a way that it's passed properly to the child process. The other thing to note is, when PowerShell is executed this way, the output of the process is *serialized objects*—the basic structure of the output is preserved—and can be passed into other commands. We'll look at this serialization in detail when we cover *remoting*—the ability to run PowerShell scripts on a remote computer—in chapter 11.

#### DESIRED STATE CONFIGURATION

Desired State Configuration (DSC) is a configuration management platform in Windows PowerShell. It enables the deployment and management of configuration data for software services and the environment on which these services run. A configuration is created using PowerShell-like syntax. The configuration is used to create a Managed Object Format (MOF) file that's passed to the remote machine on which the configuration will be applied. DSC is covered in chapter 18.

Now that we've covered the PowerShell command types, let's get back to looking at the PowerShell syntax. Notice that a lot of what we've examined this far is a bit verbose. This makes it easy to read, which is great for script maintenance, but it looks like it would be a pain to type on the command line. PowerShell addresses these two conflicting goals—readability and writeability—with the concept of *elastic syntax*. Elastic syntax allows you to expand and collapse how much you need to type to suit your purpose. We'll cover how this works in the next section.

### 1.3.4 Aliases and elastic syntax

We haven't talked about aliases yet or how they're used to achieve an elastic syntax in PowerShell. Because this concept is important in the PowerShell environment, we need to spend some time on it.

The cmdlet `Verb-Noun` syntax, although regular, is, as we noted, also verbose. You may have noticed that in some of the examples we're using commands like `dir` and `type`. The trick behind all this is aliases. The `dir` command is an alias for `Get-ChildItem`, and the `type` command is an alias for `Get-Content`. You can see this by using `Get-Command`:

```
PS> Get-Command dir

CommandType     Name
-----------     ----
Alias           dir -> Get-ChildItem
```

This tells you the command is an alias for `Get-ChildItem`. To get information about the `Get-ChildItem` command, you then do this:

```
PS> Get-Command Get-ChildItem

CommandType Name           Version Source
----------- ----           ------- ------
Cmdlet      Get-ChildItem 3.1.0.0 Microsoft.PowerShell.Management
```

To see all the information, pipe the output of `Get-Command` into `fl`. This shows you the full detailed information about this cmdlet. But wait—what's the `fl` command? Again, you can use `Get-Command` to find out:

```
PS> Get-Command fl

CommandType     Name
-----------     ----
Alias           fl -> Format-List
```

PowerShell comes with a large set of predefined aliases. Two basic categories of aliases exist: *transitional* and *convenience*. By *transitional aliases*, we mean a set of aliases that map PowerShell commands to commands that people are accustomed to using in other shells, specifically `cmd.exe` and the UNIX shells. For the `cmd.exe` user, PowerShell defines `dir`, `type`, `copy`, and so on. For the UNIX user, PowerShell defines `ls`, `cat`, `cp`, and so forth. These aliases allow a basic level of functionality for new users right away.

> **NOTE** PowerShell v6 for Linux and macOS removes these aliases to avoid confusion with native commands.

Convenience aliases are derived from the names of the cmdlets they map to. `Get-Command` becomes `gcm`, `Get-ChildItem` becomes `gci`, `Invoke-Item` becomes `ii`, and so on. For a list of the defined aliases, type `Get-Alias` at the command line. You can use the `Set-Alias` command (the alias of which is `sal`, by the way) to define your own aliases—many experienced PowerShell users create a set of one-letter aliases to cover the cmdlets they most often use at the command prompt.

> **NOTE** Aliases in PowerShell are limited to aliasing the command name only. Unlike in other systems such as Ksh, Bash, and Zsh, PowerShell aliases can't include parameters. If you need to do something more sophisticated than simple command-name translations, you'll have to use shell functions or scripts.

This is all well and good, but what does it have to do with elastics? Glad you asked! The idea is PowerShell can be terse when needed and descriptive when appropriate. The syntax is concise for simple cases and can be stretched like an elastic band for larger problems. This is important in a language that's both a command-line tool and a scripting language. Many scripts that you'll write in PowerShell will be no more than a few lines long. They will be a string of commands that you'll type on the command line and

then never use again. To be effective in this environment, the syntax needs to be concise. This is where aliases like `fl` come in—they allow you to write concise command lines. When you're scripting, though, it's best to use the long name of the command. Sooner or later, you'll have to read the script you wrote (or worse, someone else will). Would you rather read something that looks like this?

```
gcm|?{$_.parametersets.Count -gt 3}|fl name
```

or this?

```
Get-Command |
  Where-Object {$_.parametersets.count -gt 3} |
    Format-List name
```

We'd certainly rather read the latter. (As always, we'll cover the details of these examples later in the book.)

There's a second type of alias used in PowerShell: *parameter*. Unlike command aliases, which can be created by end users, parameter aliases are created by the author of a cmdlet, script, or function. (You'll see how to do this when we look at advanced function creation in chapter 7.)

A parameter alias is a shorter name for a parameter. Wait a second, earlier we said you needed enough of the parameter name to distinguish it from other command parameters. Isn't this enough for convenience and elasticity? Why do you need parameter aliases? The reason you need these aliases has to do with *script versioning*. The easiest way to understand versioning is to look at an example.

Say you have a script that calls a cmdlet `Process-Message`. This cmdlet has a parameter `-Reply`. You write your script specifying

```
Process-Message -Re
```

Run the script, and it works fine. A few months later, you install an enhanced version of the `Process-Message` command. This new version introduces a new parameter: `-Receive`. Only specifying `-Re` is no longer sufficient. If you run the old script with the new cmdlet, it will fail with an ambiguous parameter message; the script is broken.

How do you fix this with parameter aliases? The first thing to know is PowerShell always picks the parameter that exactly matches a parameter name or alias over a partial match. By providing parameter aliases, you can achieve pithiness without also making scripts subject to versioning issues. We recommend always using the full parameter name for production scripts or scripts you want to share. Readability is always more important in that scenario.

Now that we've covered the core concepts of how commands are processed, let's step back and look at PowerShell language processing overall. PowerShell has a small number of important syntactic rules you should learn. When you understand these rules, your ability to read, write, and debug PowerShell scripts will increase tremendously.

## 1.4 Parsing the PowerShell language

In this section we'll cover the details of how PowerShell scripts are parsed. Before the PowerShell interpreter can execute the commands you type, it first has to parse the command text and turn it into something the computer can execute, as shown in figure 1.4.

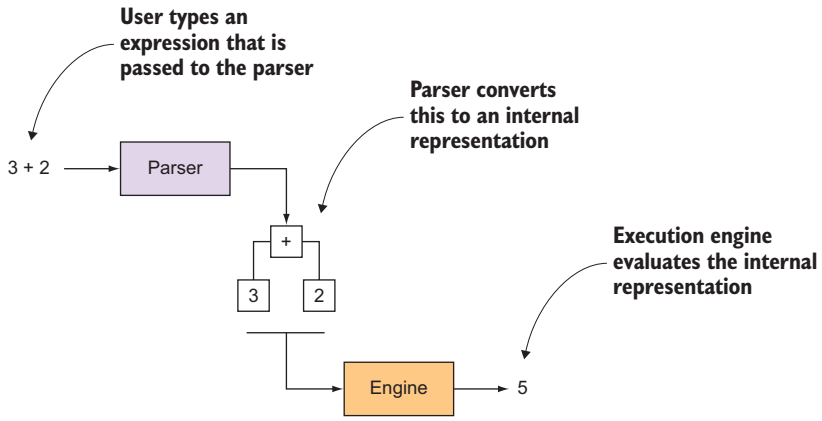



**Figure 1.4 Flow of processing in the PowerShell interpreter, where an expression is transformed and then executed to produce a result**

More formally, parsing is the process of turning human-readable source code into a form the computer understands. A piece of script text is broken up into tokens by the *tokenizer* (or *lexical analyzer*, if you want to be more technical). A token is a particular type of symbol in the programming language, such as a number, a keyword, or a variable. Once the raw text has been broken into a stream of tokens, these tokens are processed into structures in the language through syntactic analysis.

In syntactic analysis, the stream of tokens is processed according to the grammatical rules of the language. In normal programming languages, this process is straightforward—a token always has the same meaning. A sequence of digits is always a number; an expression is always an expression, and so on. For example, the sequence

```
3 + 2
```

would always be an addition expression, and "Hello world" would always be a constant string. Unfortunately, this isn't the case in shell languages. Sometimes you can't tell what a token is except through its context. In the next section, we go into more detail on why this is, and how the PowerShell interpreter parses a script.

> **NOTE** More information on this and the inner workings of PowerShell is available in the PowerShell language specification at www.microsoft.com/en-us/download/details.aspx?id=36389. The specification is currently only available up to PowerShell v3.

### 1.4.1 How PowerShell parses

For PowerShell to be successful as a shell, it can't require that everything be quoted. PowerShell would fail if it required people to continually type

```
cd ".."
```

or

```
copy "foo.txt" "bar.txt"
```

On the other hand, people have a strong idea of how expressions should work:

```
2
```

This is the number 2, not a string "2". Consequently, PowerShell has some rather complicated parsing rules, covered in the next three sections. We'll discuss how quoting is handled, the two major parsing modes, and the special rules for newlines and statement termination.

### 1.4.2 Quoting

*Quoting* is the mechanism used to turn a token that has special meaning to the PowerShell interpreter into a simple string value. For example, the `Write-Output` cmdlet has a parameter `-InputObject`. But what if you want to use the string "-InputObject" as an argument? To do this, you have to quote it by surrounding it with single or double quotes:

```
PS> Write-Output '-InputObject'
-inputobject
```

If you hadn't put the argument in quotes an error message would be produced indicating an argument to the parameter `-InputObject` is required.

PowerShell supports several forms of quoting, each with somewhat different meanings (or semantics). Putting single quotes around an entire sequence of characters causes them to be treated like a single string. This is how you deal with file paths that have spaces in them, for example. If you want to change to a directory the path of which contains spaces, you type this:

```
PS> Set-Location 'c:\program files'
PS> Get-Location
Path
----
C:\Program Files
```

When you don't use the quotes, you receive an error complaining about an unexpected parameter in the command because `c:\program` and `files` are treated as two separate tokens.

> **NOTE** Notice the error message reports the name of the cmdlet, not the alias used. This way you know what's being executed. The position message shows you the text that was entered so you can see an alias was used.

One problem with using matching quotes as shown in the previous examples is you have to remember to start the token with an opening quote. This raises an issue when you want to quote a single character. You can use the backquote (`) character to do this (the backquote is usually the upper-leftmost key, below Esc):

```
PS> Set-Location c:\program` files
PS> Get-Location
Path
----
C:\Program Files
```

The backquote, or *backtick*, as it tends to be called, has other uses that we'll explore later in this section. Now let's look at the other form of matching quote: double quotes. You'd think it works pretty much like the example with single quotes; what's the difference? In double quotes, variables are expanded. If the string contains a variable reference starting with a `$`, it will be replaced by the string representation of the value stored in the variable. Let's look at an example. First assign the string "files" to the variable `$v`:

```
PS> $v = 'files'
```

Now reference that variable in a string with double quotes:

```
PS> Set-Location "c:\program $v"
PS> Get-Location

Path
----
C:\Program Files
```

The directory change succeeded and the current directory was set as you expected.

> **NOTE** Variable expansion only occurs with double quotes. A common beginner error is to use single quotes and expect variable expansion to work.

What if you want to show the value of `$v`? To do this, you need to have expansion in one place but not in the other. This is one of those other uses we had for the backtick. It can be used to quote or escape the dollar sign in a double-quoted string to suppress expansion. Let's try it:

```
PS> Write-Output "`$v is $v"
$v is files
```

Here's one final tweak to this example—if `$v` contained spaces, you'd want to make clear what part of the output was the value. Because single quotes can contain double quotes and double quotes can contain single quotes, this is straightforward:

```
PS> Write-Output "`$v is '$v'"
$v is 'files'
```

Now, suppose you want to display the value of `$v` on another line instead of in quotes. Here's another situation where you can use the backtick as an escape character. The sequence `` `n `` in a double-quoted string will be replaced by a newline character. You can write the example with the value of `$v` on a separate line:

```
PS> "The value of `$v is:`n$v"
The value of $v is:
files
```

The list of special characters that can be generated using backtick (also called *escape*) sequences can be found using `Get-Help about_Escape_Characters`. Note that escape sequence processing, like variable expansion, is only done in double-quoted strings. In single-quoted strings, what you see is what you get. This is particularly important when writing a string to pass to a subsystem that does additional levels of quote processing.

### 1.4.3 Expression-mode and command-mode parsing

As mentioned earlier, because PowerShell is a shell, it has to deal with some parsing issues not found in other languages. PowerShell simplifies parsing considerably, trimming the number of modes down to two: expression and command.

In expression mode, the parsing is conventional: strings must be quoted, numbers are always numbers, and so on. In command mode, numbers are treated as numbers, but all other arguments are treated as strings unless they start with `$`, `@`, `'`, `"`, or `(`. When an argument begins with one of these special characters, the rest of the argument is parsed as a value expression. (There's also special treatment for leading variable references in a string, which we'll discuss later.) Table 1.1 shows examples that illustrate how items are parsed in each mode.

**Table 1.1 Parsing mode examples**

| Example command line | Parsing mode and explanation |
|---|---|
| `2+2` | Expression mode; results in `4`. |
| `Write-Output 2+2` | Command mode; results in `2+2`. |
| `$a=2+2` | Expression mode; the variable `$a` is assigned the value `4`. |
| `Write-Output (2+2)` | Expression mode; because of the parentheses, `2+2` is evaluated as an expression producing `4`. This result is then passed as an argument to the `Write-Output` cmdlet. |

**Table 1.1 Parsing mode examples *(continued)***

| Example command line | Parsing mode and explanation |
|---|---|
| `Write-Output $a` | Expression mode; produces `4`. This is ambiguous—evaluating it in either mode produces the same result. The next example shows why the default is expression mode if the argument starts with a variable. |
| `Write-Output $a.Equals(4)` | Expression mode; `$a.Equals(4)` evaluates to true and `Write-Output` writes the Boolean value `True`. This is why a variable is evaluated in expression mode by default. You want simple method and property expressions to work without parentheses. |
| `Write-Output $a/foo.txt` | Command mode; `$a/foo.txt` expands to `4/foo.txt`. This is the opposite of the previous example. Here you want it to be evaluated as a string in command mode. The interpreter first parses in expression mode and sees it's not a valid property expression, so it backs up and rescans the argument in command mode. As a result, it's treated as an expandable string. |

Notice in the `Write-Output (2+2)` case, the opening parenthesis causes the interpreter to enter a new level of interpretation where the parsing mode is once again established by the first token. This means the sequence `2+2` is parsed in expression mode, not command mode, and the result of the expression (4) is emitted. Also, the last example in the table illustrates the exception mentioned previously for a leading variable reference in a string. A variable itself is treated as an expression, but a variable followed by arbitrary text is treated as though the whole thing were in double quotes. This allows you to write

```
PS> cd $HOME/scripts
```

instead of

```
PS> cd "$HOME/scripts"
```

As mentioned earlier, quoted and unquoted strings are recognized as different tokens by the parser. This is why

```
PS> Invoke-MyCmdlet -Parm arg
```

treats `-Parm` as a parameter and

```
PS> Invoke-MyCmdlet "-Parm" arg
```

treats `"-Parm"` as an argument. There's an additional wrinkle in the parameter binding. If an unquoted parameter like `-NotAparameter` isn't a parameter on `Invoke-MyCmdlet`, it will be treated as an argument. This lets you say

```
PS> Write-Host  -this -is -a parameter
```

without requiring quoting.

This finishes our coverage of the basics of parsing modes, quoting, and commands. Commands can take arbitrary lists of arguments, so knowing when the statement ends is important. We'll cover this in the next section.

### 1.4.4 Statement termination

In PowerShell, there are two statement terminator characters: the semicolon (`;`) and (sometimes) the newline. Why is a newline a statement separator only *sometimes*? The rule is that if the previous text is a syntactically complete statement, a newline is considered to be a statement termination. If it isn't complete, the newline is treated like any other whitespace. This is how the interpreter can determine when a command or expression crosses multiple lines. For example, in the following

```
PS> 2 +
>> 2
>>
4
```

the sequence `2 +` is incomplete, so the interpreter prompts you to enter more text. (This is indicated by the nest prompt characters, `>>`.) But in the next sequence

```
PS> 2
2
PS> + 2
2
```

the number `2` by itself is a complete expression, so the interpreter goes ahead and evaluates it. Likewise, `+ 2` is a complete expression and is also evaluated (`+` in this case is treated as the unary plus operator). From this, you can see that if the newline comes after the `+` operator, the interpreter will treat the two lines as a single expression. If the newline comes before the `+` operator, it will treat the two lines as two individual expressions.

Most of the time, this mechanism works the way you expect, but sometimes you can receive some unanticipated results. Take a look at the following example:

```
PS> $b = ( 2
>> + 2 )
>>
At line:1 char:9
+ $b = ( 2
+         ~
Missing closing ')' in expression.
    + CategoryInfo          : ParserError: (:) [],
   ParentContainsErrorRecordException
    + FullyQualifiedErrorId : MissingEndParenthesisInExpression
```

> **NOTE** The example code applies to the PowerShell console. If you use ISE you'll get the error immediately after pressing the Enter key after typing the first line.

This behavior was questioned by one of the PowerShell v1 beta testers who was surprised by this result and thought there was something wrong with the interpreter, but in fact, this isn't a bug. Here's what's happening.

Consider this text:

```
PS> $b = (2 +
>> 2)
```

It's parsed as `$b = (2 + 2)` because a trailing `+` operator is only valid as part of a binary operator expression. The sequence `$b = ( 2 +` can't be a syntactically complete statement, and the newline is treated as whitespace. On the other hand, consider this text:

```
PS> $b = (2
>> + 2)
```

In this case, `2` is a syntactically complete statement, so the newline is now treated as a line terminator. In effect, the sequence is parsed like `$b = (2 ; + 2 )`—two complete statements. Because the syntax for a parenthetical expression is

```
( <expr> )
```

you get a syntax error—the interpreter is looking for a closing parenthesis as soon as it has a complete expression. Contrast this with using a subexpression instead of the parentheses alone:

```
PS> $b = $(
>> 2
>> +2
>> )
PS> $b
2
2
```

Here the expression is valid because the syntax for subexpressions is

```
$( <statementList> )
```

How do you extend a line that isn't extensible by itself? This is another situation where you can use the backtick escape character. If the last character in the line is a backtick, then the newline will be treated as a simple breaking space instead of a newline:

```
PS> Write-Output `
>> -InputObject `
>> "Hello world"
>>
Hello world
```

Finally, one thing that surprises some people is strings aren't terminated by a newline character. Strings can carry over multiple lines until a matching, closing quote is encountered:

```
PS> Write-Output "Hello
>> there
>> how are
>> you?"
>>
Hello
there
how are
you?
```

In this example, you see a string that extended across multiple lines. When that string was displayed, the newlines were preserved in the string.

The handling of end-of-line characters in PowerShell is another of the trade-offs that keeps PowerShell useful as a shell. Although the handling of end-of-line characters is a bit strange compared to non-shell languages, the overall result is easy for most people to get used to.

### 1.4.5 Comment syntax in PowerShell

Every computer language has some mechanism for annotating code with expository comments. Like many other shells and scripting languages, PowerShell comments begin with a number sign (`#`) and continue to the end of the line. The `#` character must be at the beginning of a token for it to start a comment. Here's an example that illustrates what this means (`echo` is an alias of `Write-Output`):

```
PS> echo hi#there
hi#there
```

In this example, the number sign is in the middle of the token `hi#there` and isn't treated as the starting of a comment. In the next example, there's a space before the number sign:

```
PS> echo hi #there
hi
```

Now `#` is treated as starting a comment and the following text isn't displayed. It can be preceded by characters other than a space and still start a comment. It can be preceded by any statement-terminating or expression-terminating character like a bracket, brace, or semicolon, as shown in the next couple of examples:

```
PS> (echo hi)#there
Hi

PS> echo hi;#there
hi
```

In both examples, the # symbol indicates the start of a comment.

Finally, you need to take into account whether you're in expression mode or command mode. In command mode, as shown in the next example, the + symbol is included in the token `hi+#there`:

```
PS> echo hi+#there
hi+#there
```

In expression mode, it's parsed as its own token. Now # indicates the start of a comment, and the overall expression results in an error:

```
PS> "hi"+#there
>>
At line:1 char:6
+ "hi"+#there
+      ~
You must provide a value expression following the '+' operator.
    + CategoryInfo          : ParserError: (:) [],
   ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ExpectedValueExpression
```

The # symbol is also allowed in function names:

```
PS> function hi#there { "Hi there" }
PS> hi#there
Hi there
```

The reason for allowing # in the middle of tokens was to make it easy to accommodate path providers that used # as part of their path names. People conventionally include a space before the beginning of a comment, and this doesn't appear to cause any difficulties.

### MULTILINE COMMENTS

In PowerShell v2, *multiline* comments were introduced, primarily to allow you to embed inline help text in scripts and functions. A multiline comment begins with <# and ends with #>. Here's an example:

```
<#
 This is a comment
    that spans
 multiple lines
#>
```

This type of comment need not span multiple lines; you can use this notation to add a comment preceding some code:

```
PS> <# a comment #> "Some code"
Some code
```

In this example, the line is parsed, the comment is read and ignored, and the code after the comment is executed.

One of the things this type of comment allows you to do is easily embed chunks of preformatted text in functions and scripts. The PowerShell help system takes advantage of this feature to allow functions and scripts to contain *inline documentation* in the form of special comments. These comments are automatically extracted by the help system to generate documentation for the function or script. You'll learn how the comments are used by the help subsystem in chapter 7.

Now that you have a good understanding of the basic PowerShell syntax, let's look at how commands are executed by the PowerShell execution engine. We'll start with the pipeline.

## 1.5 How the pipeline works

A pipeline is a series of commands separated by the pipe operator (|), as shown in figure 1.5. In some ways, the term *production line* better describes pipelines in PowerShell. Each command in the pipeline receives an object from the previous command, performs some operation on it, and then passes it along to the next command in the pipeline.

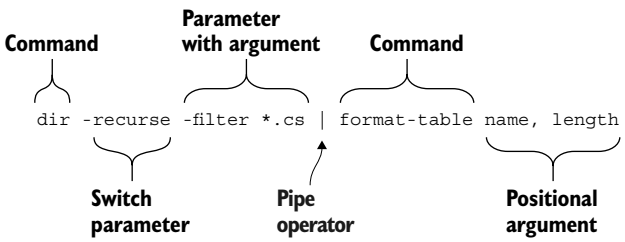



Figure 1.5 Anatomy of a pipeline

**NOTE** This, by the way, is the great PowerShell heresy. All previous shells passed strings only through the pipeline. Many people had difficulty with the notion of doing anything else. Like the character in *The Princess Bride*, they'd cry, "Inconceivable!" And we'd respond, "I do not think that word means what you think it means."

All the command categories take parameters and arguments. In

```
Get-ChildItem -Filter *.dll -Path c:\windows -Recurse
```

`-Filter` is a parameter that takes one argument, `*.dll`. The string "c:\windows" is the argument to the positional parameter `-Path`.

Next, we'll discuss the signature characteristic of pipelines: streaming behavior.

### 1.5.1 Pipelines and streaming behavior

Streaming behavior occurs when objects are processed one at a time in a pipeline. This is one of the characteristic behaviors of shell languages. In stream processing, objects are output from the pipeline as soon as they become available. In more traditional

programming environments the results are returned only when the entire result set has been generated—the first and last results are returned at the same time. In a pipelined shell, the first result is returned as soon as it's available and subsequent results return as they also become available. This flow is illustrated in figure 1.6.

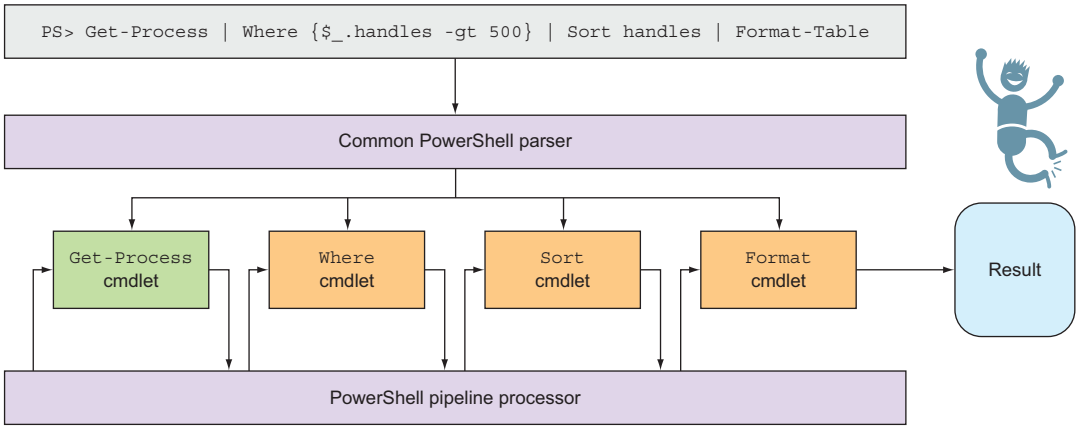



**Figure 1.6 How objects flow through a pipeline one at a time. A common parser constructs each of the command objects and then starts the pipeline processor, stepping each object through all stages of the pipeline.**

At the top of figure 1.6 you see a PowerShell command pipeline containing four commands. This command pipeline is passed to the PowerShell parser, which figures out what the commands are, what the arguments and parameters are, and how they should be bound for each command. When the parsing is complete, the pipeline processor begins to sequence the commands. First it runs the `begin` clause of each of the commands once, in sequence from first to last. After all the `begin` clauses have been run, it runs the `process` clause in the first command. If the command generates one or more objects, the pipeline processor passes these objects one at a time to the second command. If the second command also emits an object, this object is passed to the third command, and so on.

When processing reaches the end of the pipeline, any objects emitted are passed back to the PowerShell host. The host is then responsible for any further processing.

This aspect of streaming is important in an interactive shell environment, because you want to see objects as soon as they're available. The next example shows a simple pipeline that traverses through C:\Windows looking for all the DLLs with names that start with the word "system":

```
PS> Get-ChildItem -Path C:\Windows\ -recurse -filter *.dll |
where Name -match "system.*dll"
    Directory: C:\Windows\assembly\GAC_MSIL\System.Management.Automation\1.0.
  0.0__31bf3856ad364e35
```

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----       16/07/2016     12:43        3010560 System.Management.
  Automation.dll

    Directory: C:\Windows\assembly\GAC_MSIL\System.Management.Automation.
  Resources\1.0.0.0_en_31bf3856ad364e35

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----       16/07/2016     23:51         253952 System.Management.
  Automation.Resources.dll

    Directory: C:\Windows\assembly\NativeImages_v4.0.30319_32\System\08da6b66
  98b412866e6910ae9b84f363

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----       16/07/2016     12:44       10281640 System.ni.dll
```

With streaming behavior, as soon as the first file is found, it's displayed. Without streaming, you'd have to wait until the entire directory structure has been searched before you'd see any results.

In most shell environments streaming is accomplished by using separate processes for each element in the pipeline. In PowerShell, which only uses a single process (and a single thread as well by default), streaming is accomplished by splitting cmdlets into three clauses: `BeginProcessing`, `ProcessRecord`, and `EndProcessing`. In a pipeline, the `BeginProcessing` clause is run for all cmdlets in the pipeline. Then the `ProcessRecord` clause is run for the first cmdlet. If this clause produces an object, that object is passed to the `ProcessRecord` clause of the next cmdlet in the pipeline, and so on. Finally, the `EndProcessing` clauses are all run. (We cover this sequencing again in more detail in chapter 5, which is about scripts and functions, because they can also have these clauses.)

### 1.5.2 Parameters and parameter binding

Now let's talk about more of the details involved in binding parameters for commands. *Parameter binding* is the process in which values are bound to the parameters on a command. These values can come from either the command line or the pipeline. Here's an example of a parameter argument being bound from the command line:

```
PS> Write-Output 123
123
```

And here's the same example where the parameter is taken from the input object stream:

```
PS> 123 | Write-Output
123
```

The binding process is controlled by declaration information on the command itself. Parameters can have the following characteristics: they're either mandatory or optional, they have a type to which the formal argument must be convertible, and they can have attributes that allow the parameters to be bound from the pipeline. Table 1.2 describes the steps in the binding process.

**Table 1.2 Steps in the parameter binding process**

| Binding step | Description |
|---|---|
| 1. Bind all named parameters. | Find all unquoted tokens on the command line that start with a dash. If the token ends with a colon, an argument is required. If there's no colon, look at the type of the parameter and see if an argument is required. Convert the type of argument to the type required by the parameter, and bind the parameter. |
| 2. Bind all positional parameters. | If there are any arguments on the command line that haven't been used, look for unbound parameters that take positional parameters and try to bind them. |
| 3. Bind from the pipeline by value with exact match. | If the command isn't the first command in the pipeline and there are still unbound parameters that take pipeline input, try to bind to a parameter that matches the type exactly. |
| 4. If not bound, then bind from the pipe by value with conversion. | If the previous step failed, try to bind using a type conversion. |
| 5. If not bound, then bind from the pipeline by name with exact match. | If the previous step failed, look for a property on the input object that matches the name of the parameter. If the types exactly match, bind the parameter. |
| 6. If not bound, then bind from the pipeline by name with conversion. | If the input object has a property with a name that matches the name of a parameter, and the type of the property is convertible to the type of the parameter, bind the parameter. |

As you can see, this binding process is quite involved. In practice, the parameter binder almost always does what you want—that's why a sophisticated algorithm is used. Sometimes you'll need to understand the binding algorithm to get a particular behavior. PowerShell has built-in facilities for debugging the parameter-binding process that can be accessed through the `Trace-Command` cmdlet. Here's an example showing how to use this cmdlet:

```
PS> Trace-Command -Name ParameterBinding  -Option All `
-Expression { 123 | Write-Output } -PSHost
```

In this example, you're tracing the expression in the braces—that's the expression:

```
123 | Write-Output
```

This expression pipes the number 123 to the cmdlet `Write-Output`. The `Write-Output` cmdlet takes a single mandatory parameter, `-InputObject`, which allows pipeline input

by value. The tracing output is long but fairly self-explanatory, so we haven't included it here. This is something you should experiment with to see how it can help you figure out what's going on in the parameter-binding process.

And now for the final topic in this chapter: formatting and output. The formatting and output subsystem provides the magic that lets PowerShell figure out how to display the output of the commands you type.

## 1.6 Formatting and output

One of the issues people new to PowerShell face is the formatting system. As a general rule, we run commands and depend on the system to figure out how to display the results. We'll use commands such as `Format-Table` and `Format-List` to give general guidance on the shape of the display, but no specific details. Let's dig in now and see how this all works.

PowerShell is a type-based system. Types are used to determine how things are displayed, but normal objects don't usually know how to display themselves. PowerShell deals with this by including formatting information for various types of objects as part of the extended type system. This extended type system allows PowerShell to add new behaviors to existing .NET objects or extend the formatting system to cope with new types you've created. The default formatting database is stored in the PowerShell install directory, which you can get to by using the `$PSHOME` shell variable. Here's a list of the files that were included as of this writing:

```
PS> Get-ChildItem $PSHOME/*format* | Format-Table name

Name
----
Certificate.format.ps1xml
Diagnostics.Format.ps1xml
DotNetTypes.format.ps1xml
Event.Format.ps1xml
FileSystem.format.ps1xml
Help.format.ps1xml
HelpV3.format.ps1xml
PowerShellCore.format.ps1xml
PowerShellTrace.format.ps1xml
Registry.format.ps1xml
WSMan.Format.ps1xml
```

The naming convention helps users figure out the purpose of files. (The others should become clear after reading the rest of this book.) These files are XML documents that contain descriptions of how each type of object should be displayed.

> **TIP** These files are digitally signed by Microsoft. Do *not* alter them under any circumstances. You'll break things if you do.

These descriptions are fairly complex and somewhat difficult to write. It's possible for end users to add their own type descriptions, but that's beyond the scope of this

chapter. The important thing to understand is how the formatting and outputting commands work together.

### 1.6.1 Formatting cmdlets

Display of information is controlled by the type of the objects being displayed, but the user can choose the "shape" of the display by using the `Format-*` commands:

```
PS> Get-Command Format-* | Format-Table name

Name
----
Format-Hex
Format-Volume
Format-Custom
Format-List
Format-SecureBootUEFI
Format-Table
Format-Wide
```

By *shape*, we mean things such as a table or a list.

> **NOTE** `Format-Hex` is a PowerShell v5 cmdlet that is used to create displays in hexadecimal. The `Format-SecureBootUEFI` cmdlet receives certificates or hashes as input and formats the input into a content object that is returned. The `Set-SecureBootUEFI` cmdlet uses this object to update the variable. These two cmdlets are outside the scope of this section.

Here's how they work. The `Format-Table` cmdlet formats output as a series of columns displayed across your screen:

```
PS> Get-Item c:\ | Format-Table

    Directory:

Mode                 LastWriteTime      Length       Name
----                 -------------      ------       ----
d--hs-       06/06/2017     09:06                   C:\
```

PowerShell v5 automatically derives the on–screen positioning from the first few objects through the pipeline—effectively an automatic `-Autosize` parameter. This change was introduced because `-Autosize` is a blocking parameter that caused huge amounts of data to be stored in memory until all objects were available.

#### Format-Table -Autosize parameter

In PowerShell v1 through v4 `Format-Table` tries to use the maximum width of the display and guesses at how wide a particular field should be. This allows you to start seeing data as quickly as possible (streaming behavior) but doesn't always produce optimal results. You can achieve a better display by using the `-AutoSize` switch, but this requires the formatter to process every element before displaying any of them,

***(Continued)***

and this prevents streaming. PowerShell has to do this to figure out the best width to use for each field. The result in this example looks like this:

```
PS> Get-Item c:\ | Format-Table -AutoSize

    Directory:

Mode          LastWriteTime Length Name
----          ------------- ------ ----
d--hs- 06/06/2017     09:06        C:\
```

In practice, the default layout when streaming is good and you don't need to use `-Autosize`, but sometimes it can help make things more readable.

The `Format-List` command displays the elements of the objects as a list, one after the other:

```
PS> Get-Item c:\ | Format-List

    Directory:

Name           : C:\
CreationTime   : 22/08/2013 14:31:02
LastWriteTime  : 06/06/2017 09:06:56
LastAccessTime : 06/06/2017 09:06:56
```

If there's more than one object to display, they'll appear as a series of lists. This is usually the best way to display a large collection of fields that won't fit well across the screen.

The `Format-Wide` cmdlet is used when you want to display a single object property in a concise way. It will treat the screen as a series of columns for displaying the same information:

```
PS> Get-Process –Name s* | Format-Wide -Column 8 id

1372      640      516      1328     400      532      560      828
876       984      1060     1124     4
```

In this example, you want to display the process IDs of all processes with names that start with "s" in eight columns. This formatter allows for a dense display of information.

The final formatter is `Format-Custom`, which displays objects while preserving the basic structure of the object. Because most objects have a structure that contains other objects, which in turn contain other objects, this can produce extremely verbose output. Here's a small part of the output from the `Get-Item` cmdlet, displayed using `Format-Custom`:

```
PS> Get-Item c:\ | Format-Custom -Depth 1

class DirectoryInfo
{
```

```
PSPath = Microsoft.PowerShell.Core\FileSystem::C:\
PSParentPath =
PSChildName = C:\
PSDrive =
  class PSDriveInfo
  {
    CurrentLocation =
    Name = C
    Provider = Microsoft.PowerShell.Core\FileSystem
    Root = C:\
    Description = C_Drive
    Credential = System.Management.Automation.PSCredential
  }
```

The full output is considerably longer, and notice we've told it to stop walking the object structure at a depth of 1. You can imagine how verbose this output can be! Why have this cmdlet? Mostly because it's a useful debugging tool, either when you're creating your own objects or for exploring the existing objects in the .NET class libraries.

### 1.6.2 Outputter cmdlets

Now that you know how to format something, how do you output it? You don't have to worry because, by default, things are automatically sent to (can you guess?) `Out-Default`.

Note the following three examples do exactly the same thing:

```
dir | Out-Default
dir | Format-Table
dir | Format-Table | Out-Default
```

This is because the formatter knows how to get the default outputter, the default outputter knows how to find the default formatter, and the system in general knows how to find the defaults for both. The Möbius strip of subsystems!

As with the formatters, there are several outputter cmdlets available in PowerShell out of the box. You can use the `Get-Command` command to find them:

```
PS> Get-Command Out-* | Format-Wide -Column 3

Out-Default       Out-File       Out-GridView
Out-Host          Out-Null       Out-Printer
Out-String
```

Here there's a somewhat broader range of choices. We've already talked about `Out-Default`. The next one we'll talk about is `Out-Null`. This is a simple outputter; anything sent to `Out-Null` is discarded. This is useful when you don't care about the output for a command; you want the side effect of running the command.

> **NOTE** Piping to `Out-Null` is the equivalent to redirecting to `$null` but invokes the pipeline and can be up to forty times slower than redirecting to `$null`.

Next, we have `Out-File`. Instead of sending the output to the screen, this command sends it to a file. (This command is also used by I/O redirection when doing output to

a file.) In addition to writing the formatted output, `Out-File` has several flags that control how the output is written. The flags include the ability to append to a file instead of replacing it, to force writing to read-only files, and to choose the output encodings for the file. This last item is the trickiest. You can choose from a number of text encodings supported by Windows. Here's a trick—enter the command with an encoding you know doesn't exist:

```
PS> Out-File -encoding blah
Out-File : Cannot validate argument on parameter 'Encoding'. The argument
"blah" does not belong to the set "unknown,string,unicode,bigendianunicode,ut
f8,utf7,utf32,ascii,default,oem" specified by the ValidateSet attribute.
Supply an argument that is in the set and then try the command again.
At line:1 char:20
+ Out-File -encoding blah
+                    ~~~~
    + CategoryInfo          : InvalidData: (:) [Out-File],
   ParameterBindingValidationException
    + FullyQualifiedErrorId : ParameterArgumentValidationError,Microsoft.
   PowerShell.Commands.OutFileCommand
```

You can see in the error message that all the valid encoding names are displayed.

> **NOTE** Tab completion can be used to cycle through the valid encodings. Type `Out-File -Encoding` and then keep pressing the tab key to view the options. Tab completion works with cmdlet names, parameters, and values where there's a predefined set of acceptable values.

If you don't understand what these encodings are, don't worry about it, and let the system use its default value.

> **NOTE** Where you're likely to run into problems with output encoding (or input encoding for that matter) is when you're creating files that are going to be read by another program. These programs may have limitations on what encodings they can handle, particularly older programs. To find out more about file encodings, search for "file encodings" on http://msdn.microsoft.com. Microsoft Developer's Network (MSDN) contains a wealth of information on this topic. Chapter 5 also contains additional information about working with file encodings in PowerShell.

The `Out-Printer` cmdlet doesn't need much additional explanation; it routes its text-only output to the default printer instead of to a file or to the screen.

The `Out-Host` cmdlet is a bit more interesting—it sends its output back to the host. This has to do with the separation in PowerShell between the interpreter or engine, and the application that hosts that engine. The host application has to implement a special set of interfaces to allow `Out-Host` to render its output properly. (We see this used in PowerShell v2 to v5, which include two hosts: the console host and the Integrated Scripting Environment (ISE).)

> **NOTE** `Out-Default` delegates the work of outputting to the screen to `Out-Host`.

The last output cmdlet to discuss is `Out-String`. This one's a bit different. All the other cmdlets terminate the pipeline. The `Out-String` cmdlet formats its input and sends it as a string to the next cmdlet in the pipeline. Note we said *string*, not *strings*. By default, it sends the entire output as a single string. This isn't always the most desirable behavior—a collection of lines is usually more useful—but at least once you have the string, you can manipulate it into the form you want. If you do want the output as a series of strings, use the `-Stream` switch parameter. When you specify this parameter, the output will be broken into lines and streamed one at a time.

Note this cmdlet runs somewhat counter to the philosophy of PowerShell; once you've rendered the object to a string, you've lost its structure. The main reason for including this cmdlet is for interoperation with existing APIs and external commands that expect to deal with strings. If you find yourself using `Out-String` a lot in your scripts, stop and think if it's the best way to attack the problem.

PowerShell v2 introduced one additional output command: `Out-GridView`. As you might guess from the name, this command displays the output in a grid, but rather than rendering the output in the current console window, a new window is opened with the output displayed using a sophisticated grid control (see figure 1.7).

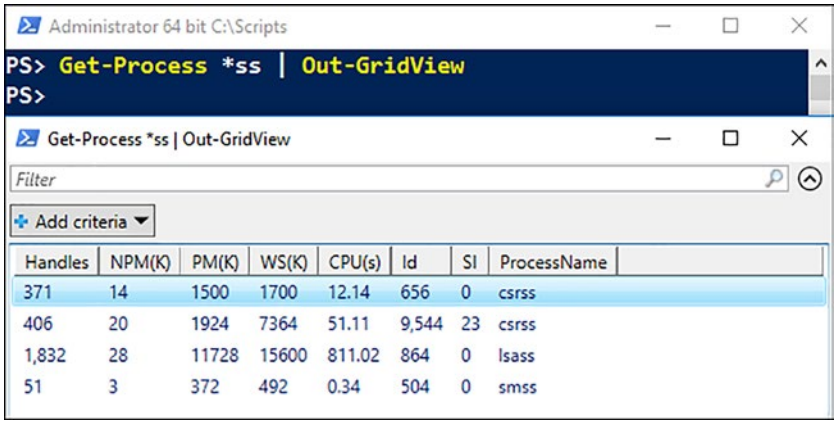



**Figure 1.7 Displaying output with `Out-GridView`**

The underlying grid control used by `Out-GridView` has all the features you'd expect from a modern Windows interface: columns can be reordered by dragging and dropping them, and the output can be sorted by clicking a column head. This control also introduces sophisticated filtering capabilities. This filtering allows you to drill into a dataset without having to rerun the command.

That's it for the basics: commands, parameters, pipelines, parsing, and presentation. You should now have a sufficient foundation to start moving on to more advanced topics in PowerShell.

## 1.7 Summary

- PowerShell is Microsoft's command-line/scripting environment that's at the center of Microsoft server and application management technologies. Microsoft's most important server products, including Exchange, Active Directory, and SQL Server, now use PowerShell as their management layer.
- PowerShell incorporates object-oriented concepts into a command-line shell using the .NET object model as the base for its type system, but can also access other object types like WMI.
- Shell operations like navigation and file manipulation in PowerShell are similar to what you're used to in other shells.
- Use the `Get-Help` command to get help when working with PowerShell.
- PowerShell has a full range of calculation, scripting, and text-processing capabilities.
- PowerShell supports a comprehensive set of remoting features to allow you to do scripted automation of large collections of computers.
- PowerShell has a number of command types, including cmdlets, functions, script commands, and native commands, each with slightly different characteristics.
- PowerShell supports an elastic syntax—concise on the command line and complete in scripts. Aliases are used to facilitate elastic syntax.
- PowerShell parses scripts in two modes—expression mode and command mode—which is a critical point to appreciate when using PowerShell.
- The PowerShell escape character is a backtick (`), not a backslash.
- PowerShell supports both double quotes and single quotes; variable and expression expansion is done in double quotes, not in single quotes.
- Line termination is handled specially in PowerShell because it's a command language.
- PowerShell has two types of comments: line comments that begin with `#` and block comments that start with `<#` and end with `#>`. The block comment notation was introduced in PowerShell v2 with the intent of supporting inline documentation for scripts and functions.
- PowerShell uses a sophisticated formatting and outputting system to determine how to render objects without requiring detailed input from the user.

Now that you have the basics, we'll start digging into the details starting in the next chapter with how PowerShell works with types.

WINDOWS ADMINISTRATION

# Windows PowerShell IN ACTION

THIRD EDITION

Bruce Payette • Richard Siddaway

In 2006, Windows PowerShell reinvented the way administrators and developers interact with Windows. Today, PowerShell is required knowledge for Windows admins and devs. This powerful, dynamic language provides command-line control of the Windows OS and most Windows servers, such as Exchange and SCCM. And because it's a first-class .NET language, you can build amazing shell scripts and tools without reaching for VB or C#.

**Windows PowerShell in Action, Third Edition** is the definitive guide to PowerShell, now revised to cover PowerShell 6. Written by language designer Bruce Payette and MVP Richard Siddaway, this rich book offers a crystal-clear introduction to the language along with its essential everyday use cases. Beyond the basics, you'll find detailed examples on deep topics like performance, module architecture, and parallel execution.

## What's Inside

- The best end-to-end coverage of PowerShell available
- Updated with coverage of PowerShell v6
- PowerShell workflows
- PowerShell classes
- Writing modules and scripts
- Desired State Configuration
- Programming APIs and pipelines

Written for intermediate-level developers and administrators.

**Bruce Payette** is codesigner and principal author of the PowerShell language. **Richard Siddaway** is a longtime PowerShell MVP, author, speaker, and blogger.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/windows-powershell-in-action-third-edition

MANNING $59.99 / Can $79.99 [INCLUDING eBOOK]


"This comprehensive guide to PowerShell just gets better with every revision!"
—Wayne Boaz, Nike

"Excellent coverage of the new features in PowerShell. Recommended for all levels of users."
—Lincoln Bovee, Proto Labs

"Deep technical discussions of the inner workings of PowerShell. Many useful examples. Up to date!"
—Dr. Edgar Knapp
ISIS Papyrus Europe

"If you're serious about PowerShell, you need to read this book. Seriously: Read this book!"
—Stephen Byrne, Dell

ISBN-13: 978-1-63343-029-7
ISBN-10: 1-63343-029-4
55999
9 781633 430297