SAMPLE CHAPTER

Learn WINDOWS POWERSHELL IN A MONTH OF LUNCHES

DON JONES

MANNING

Learn Windows PowerShell in a Month of Lunches

by Don Jones

Chapter 4

Copyright 2011 Manning Publications

brief contents

4 The pipeline: connecting commands

In chapter 2, you saw that running commands in PowerShell is basically the same as running commands in any other shell: you type a command name, give it some parameters, and hit Return. What makes PowerShell so special isn't the way it runs commands, but rather the way it allows multiple commands to be connected to each other in powerful, one-line sequences.

4.1 Connect one command to another: less work for you!

PowerShell connects commands to each other in something called a *pipeline*. The pipeline is simply a way for one command to pass, or pipe, its output to another command, so that the second command has something to work with.

You've already seen this in action when you run something like `Dir | More`. You're piping the output of the `Dir` command into the `More` command; the `More` command takes that directory listing and displays it one page at a time. PowerShell takes that same piping concept and extends it to much greater effect. In fact, PowerShell's use of a pipeline may seem similar, at first, to how Unix and Linux shells work. Don't be fooled, though. As you'll come to realize over the next few chapters, PowerShell's pipeline implementation is much richer and more modern.

4.2 Exporting to a CSV or XML file

Run a simple command. Here are a few suggestions:

- `Get-Process` (or `Ps`)
- `Get-Service` (or `Gsv`)
- `Get-EventLog Security -newest 100`

I chose these because they're easy, straightforward commands; in parentheses, I've given you aliases for `Get-Process` and `Get-Service`. For `Get-EventLog`, I also specified its mandatory parameter as well as the `-newest` parameter (so the command wouldn't take too long to execute).

> **TRY IT NOW** Go ahead and choose one of these commands to work with. I'll use `Get-Process` for the following examples; you can stick with one of these, or switch between them to see the differences in the results.

What do you see? When I run `Get-Process`, a table (shown in figure 4.1) with several columns of information appears on the screen.

It's great to have that information on the screen, but that isn't all I might want to do with the information. For example, if I wanted to make some charts and graphs of memory and CPU utilization, I might want to export the information into a CSV (comma-separated values) file that could be read into an application like Microsoft Excel.

That's where the pipeline, and a second command, come in handy:

```
Get-Process | Export-CSV procs.csv
```

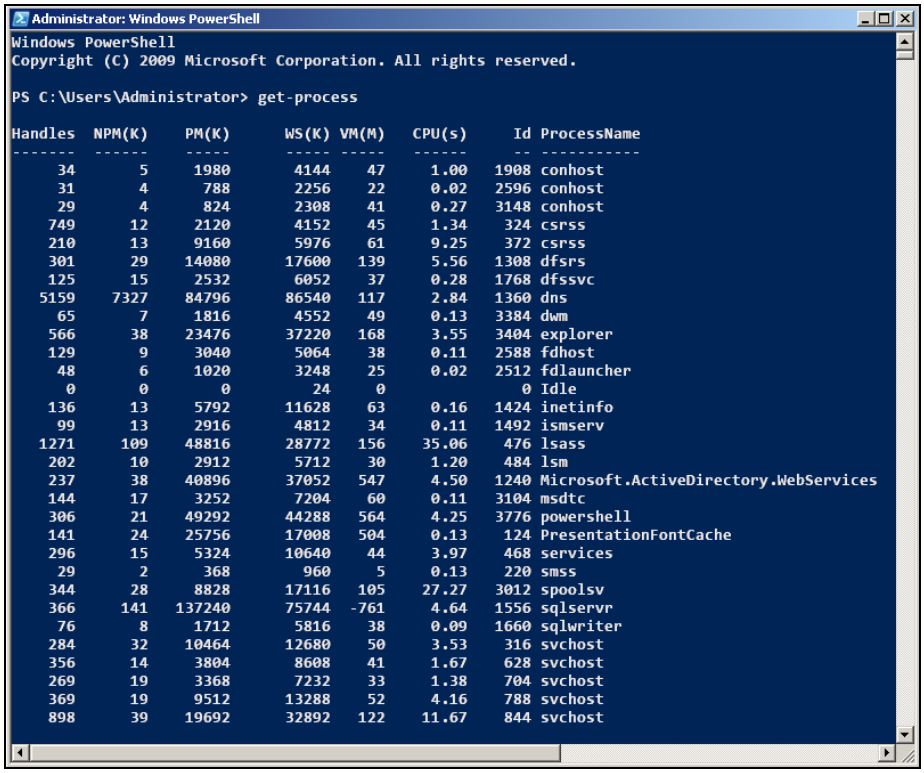

Figure 4.1 The output of `Get-Process` is a table with several columns of information.

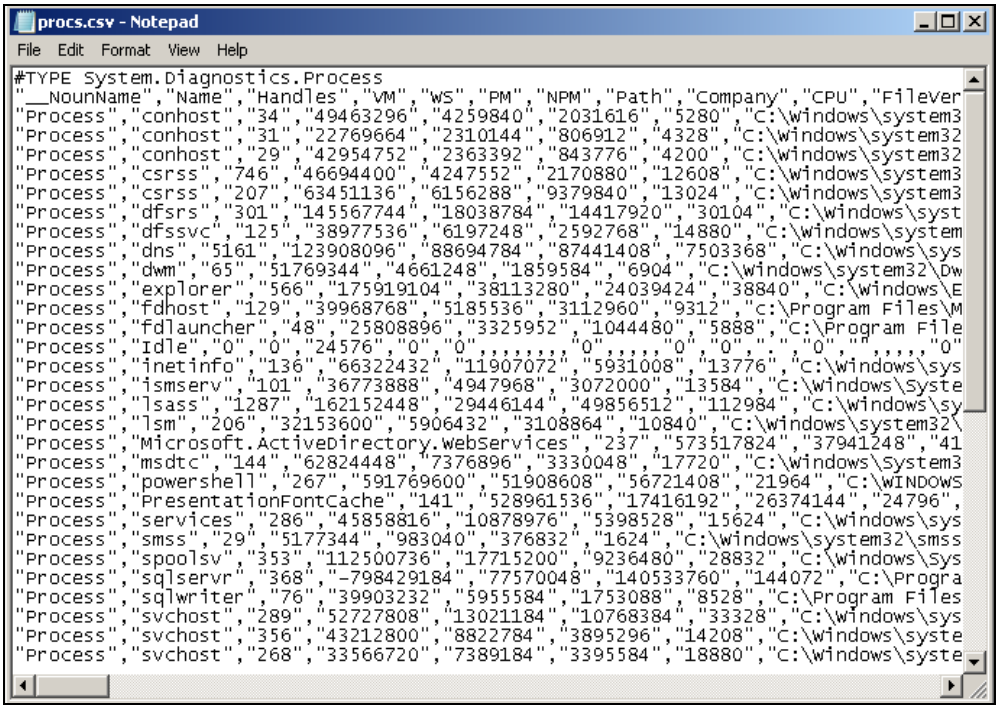

Figure 4.2 Viewing the exported CSV file in Windows Notepad

Just like piping `Dir` to `More`, I've piped my processes to `Export-CSV`. That second cmdlet has a mandatory positional parameter that I've used to specify the output filename. Because `Export-CSV` is a native PowerShell cmdlet, it knows how to translate the table normally generated by `Get-Process` into a normal CSV file.

Go ahead and open the file in Windows Notepad to see the results, as shown in figure 4.2:

```
Notepad procs.csv
```

The first line of the file will be a comment, preceded by a # sign, and it identifies the kind of information that's included in the file. In my example, it's `System.Diagnostics.Process`, which is the under-the-hood name that Windows uses to identify the information related to a running process. The second line will be column headings, and the subsequent lines will list the information for the various processes running on the computer.

You can pipe the output of almost any `Get-` cmdlet to `Export-CSV` and get excellent results. You may also notice that the CSV file contains a great deal more information than what is normally shown on the screen. That's deliberate. The shell knows it

couldn't possibly fit all of that information on the screen, so it uses a configuration file, supplied by Microsoft, to select the most important information for on-screen display. In later chapters, I'll show you how to override that configuration to display whatever you want.

Once the information is saved into a CSV file, you could easily email it to a colleague and ask them to view it from within PowerShell. They'd simply import the file:

```
Import-CSV procs.csv
```

The shell would read in the CSV file and display the process information. It wouldn't be based on live information, of course, but it would be a snapshot from the exact point in time when you created the CSV file.

What if CSV files aren't what you need? PowerShell also has an `Export-CliXML` cmdlet, which creates a generic command-line interface (CLI) Extensible Markup Language (XML) file. CliXML is unique to PowerShell, but it can be read by any program capable of understanding XML. There's also a matching `Import-CliXML` cmdlet. Both cmdlets, like `Import-CSV` and `Export-CSV`, expect a filename as a mandatory parameter.

> **TRY IT NOW** Try exporting something, such as services, processes, or event log entries, to a CliXML file. Make sure you can re-import the file, and try opening the resulting file in Notepad and Internet Explorer to see how each of those applications displays the information.

Does PowerShell include any other import or export commands? You could find out by using the `Get-Command` cmdlet and specifying a `-verb` parameter with either `Import` or `Export`.

> **TRY IT NOW** See if PowerShell comes with any other import or export cmdlets. You may want to repeat this check after you load new commands into the shell, which is something you'll do in the next chapter.

Both CSV and CliXML files can be useful for persisting snapshots of information, sharing those snapshots with others, and reviewing those snapshots at a later time. In fact, let's look at one more cmdlet that has a great way of using those snapshots: `Compare-Object`. It has an alias, `Diff`, which I'll use.

First, run `help diff` and read the help for this cmdlet. There are three parameters in particular that I want you to pay attention to: `-ReferenceObject`, `-DifferenceObject`, and `-Property`.

`Diff` is designed to take two sets of information and compare them to each other. For example, imagine that you ran `Get-Process` on two different computers that were sitting side by side. The computer that's configured just the way you want is on the left and is the *reference computer*. The computer on the right might be exactly the same, or it might be somewhat different; it's the *difference computer*. After running the command

on each, you'll be staring at two tables of information, and your job is to figure out if there are any differences between the two.

Because these are processes that you're looking at, you're always going to see differences in things like CPU and memory utilization numbers, so we'll ignore those columns. In fact, just focus on the Name column, because we really want to see if the difference computer contains any additional, or any fewer, processes than the reference computer. It might take you a while to compare all the process names from both tables, but you don't have to—that's exactly what `Diff` will do for you.

Let's say you sit down at the reference computer and run this:

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

I prefer CliXML over CSV for comparisons like this, because CliXML can hold more information than a flat CSV file. You then transport that XML file over to the difference computer, and run this:

```
Diff -reference (Import-CliXML reference.xml)
➥ -difference (Get-Process) -property Name
```

This is a bit tricky, so I'll walk you through what's happening:

- Just like in math, parentheses in PowerShell control the order of execution. In this example, they force `Import-CliXML` and `Get-Process` to run before `Diff` runs. The output from `Import-CLI` is fed to the `-reference` parameter, and the output from `Get-Process` is fed to the `-difference` parameter.

  Actually, those parameter names are `-referenceObject` and `-differenceObject`; keep in mind that you can abbreviate parameter names by typing just enough of their names for the shell to be able to figure out which one you meant. In this case, `-reference` and `-difference` are more than enough to uniquely identify these parameters. I probably could have shortened them even further to something like `-ref` and `-diff`, and the command would still have worked.
- Rather than comparing the two complete tables, `Diff` focuses on the `Name`, because I gave it the `-property` parameter. If I hadn't, it would think that every process is different because the values of columns like VM, CPU, and PM are always going to be different.
- The result will be a table telling you what's different. Every process that's in the reference set, but not in the difference set, will have a `<=` indicator (indicating that the process is only present on the left side). If a process is on the difference computer but not the reference computer, it'll have a `=>` indicator instead. Processes that match across both sets won't be included in the `Diff` output.

TRY IT NOW Go ahead and try this. If you don't have two computers, start by exporting your current processes to a CliXML file as I've shown above. Then, start some additional processes like Notepad, Windows Paint, Solitaire, or

whatever. Your computer will then be the difference computer (on the right), whereas the CliXML file will still be the reference set (on the left).

Here's the output from my test:

```
PS C:\> diff -reference (import-clixml reference.xml) -difference (get
-process) -property name

name                                SideIndicator
----                                -------------
calc                                =>
mspaint                             =>
notepad                             =>
conhost                             <=
powershell_ise                      <=
```

This is a really useful management trick. If you think of those reference CliXML files as configuration baselines, you can compare any current computer to that baseline and get a difference report. Throughout this book, you'll discover more cmdlets that can retrieve management information, all of which can be piped into a CliXML file to become a baseline. You can quickly build a collection of baseline files for services, processes, operating system configuration, users and groups, and much more, and then use those at any time to compare the current state of a system to its baseline.

> **TRY IT NOW** Just for fun, try running the `Diff` command again, but leave off the `-property` parameter entirely. See the results? Every single process is listed, because values like PM, VM, and so forth have all changed, even though they're the same processes. The output also isn't as useful, because it simply displays the process's type name and process name.

By the way, you should know that `Diff` generally doesn't do well at comparing text files. Although other operating systems and shells have a `Diff` command that's explicitly intended for comparing text files, PowerShell's `Diff` command works very differently. You'll see just how differently in this chapter's concluding lab.

> **NOTE** If it seems like you're using `Get-Process`, `Get-Service`, and `Get-EventLog` a lot, well, that's on purpose. I'm guaranteed that you have access to those cmdlets because they're native to PowerShell and don't require an add-in like Exchange or SharePoint. That said, the skills you're learning will apply to every cmdlet you ever need to run, including those that ship with Exchange, SharePoint, SQL Server, and other server products. Chapter 26 will go into that idea in more detail, but for now, focus on *how* to use these cmdlets rather than what the cmdlets are accomplishing. I'll work in some other representative cmdlets at the right time.

4.3 Piping to a file or printer

Whenever you have nicely formatted output—like the tables generated by `Get-Service` or `Get-Process`—you may want to preserve that in a file, or even on paper. Normally,

cmdlet output is directed to the screen, which PowerShell refers to as the *Host*. You can change where that output goes. In fact, I've already showed you one way to do so:

```
Dir > DirectoryList.txt
```

That's a shortcut added to PowerShell to provide syntactic compatibility with the older Cmd.exe shell. In reality, when you run that command, here's what PowerShell does under the hood:

```
Dir | Out-File DirectoryList.txt
```

You can run that same command on your own, instead of using the `>` syntax. Why would you do so? `Out-File` also provides additional parameters that let you specify alternative character encodings (such as UTF8 or Unicode), append content to an existing file, and so forth. By default, the files created by `Out-File` are 80 columns wide, so sometimes PowerShell might alter command output to fit within 80 characters. That alteration might make the file's contents appear different than when you run the same command on the screen. Read its help file and see if you can spot a parameter of `Out-File` that would let you change the output file width to something other than 80 characters.

> **TRY IT NOW** Don't look here—open up that help file and see what you can find. I guarantee you'll spot the right parameter in a few moments.

PowerShell has a variety of `Out-` cmdlets. One is called `Out-Default`, and that's the one the shell uses when you don't specify a different `Out-` cmdlet. If you run this,

```
Dir
```

you're technically running this,

```
Dir | Out-Default
```

even if you don't realize it. `Out-Default` does nothing more than direct content to `Out-Host`, so you're really running this,

```
Dir | Out-Default | Out-Host
```

without realizing it. `Out-Host` is what handles getting information displayed on the screen. What other `Out-` cmdlets can you find?

> **TRY IT NOW** See what other `Out-` cmdlets you can discover. One way would be to use the `Help` command, using wildcards, such as `Help Out*`. Another would be to use `Get-Command` the same way, such as `Get-Command Out*`. Or, you could specify the `-verb` parameter: `Get-Command -verb Out`. What do you come up with?

`Out-Printer` is probably one of the most useful of the remaining `Out-` cmdlets. `Out-GridView` is also neat; it does require, however, that you have Microsoft .NET Framework v3.5 and the Windows PowerShell ISE installed, which isn't the case by default on server operating systems.

If you do have those installed, try running `Get-Service | Out-GridView` to see what happens. `Out-Null` and `Out-String` have specific uses that we won't get into right now, but you're welcome to read their help files and look at the examples included in those files.

4.4 Converting to HTML

Want to produce HTML reports? Easy: pipe your command to `ConvertTo-HTML`. This command produces well-formed, generic HTML that will display in any web browser. It's plain-looking, but you can reference a Cascading Style Sheet (CSS) to specify prettier formatting if desired. Notice that this doesn't require a filename:

```
Get-Service | ConvertTo-HTML
```

TRY IT NOW Make sure you run that command yourself—I want you to see what it does before you proceed.

In the PowerShell world, the verb `Export` implies that you're taking data, converting it to some other format, and saving that other format in some kind of storage, such as a file. The verb `ConvertTo` implies only a portion of that process: the conversion to a different format, but not saving it into a file. So when you ran the preceding command, you got a screen full of HTML, which probably isn't what you want. Stop for a second: can you think of how you'd get that HTML into a text file on disk?

TRY IT NOW If you can think of a way, go ahead and try it before you read on.

This command would do the trick:

```
Get-Service | ConvertTo-HTML | Out-File services.html
```

See how connecting more and more commands allows you to have increasingly powerful command lines? Each command handles a single step in the process, and the entire command line as a whole accomplishes a useful task.

PowerShell ships with other `ConvertTo-` cmdlets, including `ConvertTo-CSV` and `ConvertTo-XML`. As with `ConvertTo-HTML`, these don't create a file on disk; they translate command output into CSV or XML, respectively. You could pipe that converted output to `Out-File` to then save it to disk, although it would be shorter to use `Export-CSV` or `Export-CliXML`, because those do both the conversion and the saving.

Above and beyond

Time for a bit more useless background information, although, in this case, it's the answer to a question that a lot of students often ask me: why would Microsoft provide both `Export-CSV` and `ConvertTo-CSV`, as well as two nearly identical cmdlets for XML? In certain advanced scenarios, you might not want to save the data to a file on disk. For example, you might want to convert data to XML and then transmit it to a web service, or some other destination. By having distinct `ConvertTo-` cmdlets that don't save to a file, you have the flexibility of doing whatever you want.

4.5 Using cmdlets to kill processes and stop services

Exporting and converting aren't the only reasons you might want to connect two commands together. For example, consider—but *please do not run*—this command:

```
Get-Process | Stop-Process
```

Can you imagine what that command would do? I'll tell you: crash your computer. It would retrieve every process and then start trying to end each one of them. It would get to a critical process, like the Local Security Authority, and your computer would probably crash with the famous Blue Screen of Death (BSOD). If you're running PowerShell inside of a virtual machine and want to have a little fun, go ahead and try running that command.

The point is that cmdlets with the same noun (in this case, `Process`) can often pass information between each other. Typically, you would specify the name of a specific process rather than trying to stop them all:

```
Get-Process -name Notepad | Stop-Process
```

Services offer something similar: the output from `Get-Service` can be piped to cmdlets like `Stop-Service`, `Start-Service`, `Set-Service`, and so forth. As you might expect, there are some specific rules about which commands can connect to each other. For example, if you look at a command sequence like `Get-ADUser | New-SQLDatabase`, you would probably not expect it to do anything sensible (although it might well do something nonsensical). In chapter 7, we'll dive into the rules that govern how commands can connect to each other.

There is one more thing I'd like you to know about cmdlets like `Stop-Service` and `Stop-Process`. These cmdlets modify the system in some fashion, and all cmdlets that modify the system have an internally defined *impact level*. This impact level is set by the cmdlet's creator, and it can't be changed. The shell has a corresponding `$ConfirmPreference` setting, which is set to `High` by default. You can see your shell's setting by typing the setting name, like this:

```
PS C:\> $confirmpreference
High
```

Here's how it works: When a cmdlet's internal impact level is equal to or higher than the shell's `$ConfirmPreference` setting, the shell will automatically ask, "Are you sure?" when the cmdlet does whatever it's trying to do. In fact, if you tried the crash-your-computer command, earlier, you probably were asked, "Are you sure?" for each process. When a cmdlet's internal impact level is less than the shell's `$ConfirmPreference`, you don't automatically get the "Are you sure?" prompt.

You can, however, force the shell to ask you if you're sure:

```
Get-Service | Stop-Service -confirm
```

Just add the `-confirm` parameter to the cmdlet. This should be supported by any cmdlet that makes some kind of change to the system, and it'll show up in the help file for the cmdlet if it's supported.

A similar parameter is `-whatif`. This is supported by any cmdlet that supports `-confirm`. The `-whatif` parameter isn't triggered by default, but you can specify it whenever you want to:

```
PS C:\> get-process | stop-process -whatif
What if: Performing operation "Stop-Process" on Target "conhost (1920)
".
What if: Performing operation "Stop-Process" on Target "conhost (1960)
".
What if: Performing operation "Stop-Process" on Target "conhost (2460)
".
What if: Performing operation "Stop-Process" on Target "csrss (316)".
```

It tells you what the cmdlet would have done, without actually letting the cmdlet do it. It's a useful way to preview what a potentially dangerous cmdlet would have done to your computer, to make certain that you want to do that.

4.6 Lab

I've kept this chapter's text a bit shorter because some of the examples I showed you probably took a bit longer to complete, and because I want you to spend a bit more time completing the following hands-on exercises. If you haven't already completed all of the "Try it now" tasks in the chapter, I strongly recommend that you do so before tackling these tasks:

1. Create a CliXML reference file for the services on your computer. Then, change the status of some non-essential service like BITS (stop it if it's already started; start it if it's stopped on your computer). Finally, use `Diff` to compare the reference CliXML file to the current state of your computer's services. You'll need to specify more than the `Name` property for the comparison—does the `-property` parameter of `Diff` accept multiple values? How would you specify those multiple values?
2. Create two similar, but different, text files. Try comparing them using `Diff`. To do so, run something like this: `Diff -reference (Get-Content File1.txt) -difference (Get-Content File2.txt)`. If the files have only one line of text that's different, the command should work. If you add a bunch of lines to one file, the command may stop working. Try experimenting with the `Diff` command's `-syncWindow` parameter to see if you can get the command working again.
3. What happens if you run `Get-Service | Export-CSV services.csv | Out-File` from the console? Why does that happen?
4. Apart from getting one or more services and piping them to `Stop-Service`, what other means does `Stop-Service` provide for you to specify the service or services you want to stop? Is it possible to stop a service without using `Get-Service` at all?

5 What if you wanted to create a pipe-delimited file instead of a comma-separated file? You would still use the `Export-CSV` command, but what parameters would you specify?

6 Is there a way to eliminate the `#` comment line from the top of an exported CSV file? That line normally contains type information, but what if you wanted to omit that from a particular file?

7 `Export-CliXML` and `Export-CSV` both modify the system, because they can create and overwrite files. What parameter would prevent them from overwriting an existing file? What parameter would ask you if you were sure before proceeding to write the output file?

8 Windows maintains several regional settings, which include a default list separator. On U.S. systems, that separator is a comma. How can you tell `Export-CSV` to use the system's default separator, rather than a comma?

WINDOWS ADMINISTRATION

Learn WINDOWS POWERSHELL IN A MONTH OF LUNCHES

DON JONES

Free ebook
SEE INSERT

Windows has so many control panels, consoles, APIs, and wizards it's really hard to keep track of all the locations and settings you'll need. PowerShell is a godsend: it provides a single, unified administrative command line. It accepts and executes commands immediately. And it has in-built language features that will let you write scripts to control any Windows component, including servers like Exchange, IIS, and SharePoint.

Learn Windows PowerShell in a Month of Lunches is a newly designed tutorial for system administrators. Just set aside one hour a day—lunchtime would be perfect—for a month, and you'll be automating administrative tasks in a hurry. Author Don Jones combines his in-the-trenches experience with a unique teaching style to help you master the effective parts of PowerShell quickly and painlessly.

What's Inside

- Learn PowerShell 2—no experience required!
- Concise lessons for busy administrators
- Practical examples and techniques in every lesson

About the Author

Don Jones is a PowerShell MVP, speaker, and trainer. He developed the Microsoft PowerShell courseware and has taught PowerShell to more than 20,000 IT pros. Don writes the PowerShell column for TechNet Magazine and blogs for WindowsITPro.com.

"A seminal guide to PowerShell. Highly recommended."
—Ray Booysen, BNP Paribas

"The book I wish I'd had when I started PowerShell."
—Richard Siddaway, Serco

"Tons of useful exercises allow powerful hands-on learning."
—Chuck Durfee
Graebel Companies

"Whether a beginner or an intermediate, you'll need *no other* book."
—David Moravec
PowerShell.cz

For access to the book's forum and a free ebook for owners of this book, go to manning.com/LearnWindowsPowerShellinaMonthofLunches

ISBN-13: 978-1-617290213
ISBN-10: 1-617290211
54499
9 781617 290213

MANNING US $44.99/CAN $51.99 [INCLUDING EBOOK]