

Windows PowerShell: TFM™
Sample Chapters

Windows PowerShell™: TFM®

Don Jones
Jeffery Hicks

SAPIEN Press
Napa, California

Windows PowerShell: TFM™
Sample Chapters

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and SAPIEN Press was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Windows is a registered trademark, and Windows PowerShell is a trademark, of Microsoft Corporation in the United States and other countries.

The author and publisher have taken care in the publication of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or software programs contained herein.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

SAPIEN Press
1-866-PRIMALS
sales@sapien.com

Visit SAPIEN Press on the Web : www.sapienpress.com

Library of Congress Cataloging-in-Publication Data

Jones, Don; Hicks, Jeffery

Microsoft PowerShell: TFM / Don Jones and Jeffery Hicks

p. cm.

ISBN 0-9776597-2-0 (pbk. : alk. Paper)

1. Microsoft Windows (computer file)
2. Operating systems (Computers)
3. PowerShell (Computer program language) I. Title

Copyright ©2006 by SAPIEN Technologies, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior express written consent of the publisher. Printed in the United States of America.

For information on obtaining permission for use of material from this work, please submit a written request to:

SAPIEN Technologies, Inc.
Rights and Contracts Department
3212 Jefferson St #288
Napa, CA 94558
Fax: 707-252-8700

1st Printing

Foreword

In writing a book for Windows PowerShell, my co-author and I had some difficult decisions to make. PowerShell is a terrifically powerful environment; however, with all that power comes a lot of complexity. It's fascinating, but it's also easy to find yourself slipping out of the real easy-administrative-scripting world and into a more hardcore .NET Framework developer's world. We decided to keep our audience, the Windows administrator, firmly in mind. After all, that's where both Jeffery and I come from, and we know what a pragmatic, practical bunch administrators are. Typically, administrators just want a tool that'll let them get their jobs done with the minimum amount of additional fuss or effort.

That said, administrators' specific needs vary pretty widely. So we decided to stick with the good old "80/20" rule: Cover the information that 80% of the audience will use 80% of the time, and consider everything else to be "out of scope." We did this not out of a desire to "dumb down" the book or provide a less-comprehensive work, but rather to keep the book focused on what *most* administrators will need to use *most* of the time. Doing so keeps the discussion free of distractions which *most* readers won't find useful, allowing *most* readers to learn how to use PowerShell as quickly as possible.

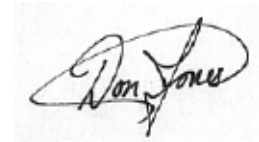
And *learning* is really the key goal of this book: While we've included a good deal of reference material, it's not intended as a reference, but rather as a learning tool. So whenever we needed to make a tradeoff between "easy, clear instruction" and "comprehensive coverage," we've generally erred on the side of ease and clarity. As more Windows administrators become familiar with, and proficient in, PowerShell we'll likely get feedback about features or techniques that we didn't include. We want that feedback! In fact, visit www.SAPIENPress.com to give us that feedback when it occurs to you; the time will doubtless arrive for an "advanced" PowerShell title that's more comprehensive and doesn't need to worry about teaching the basics, and your feedback will help craft that title.

In the meantime, we hope you'll find *this* title useful. We've tried to focus on the core PowerShell techniques that you'll get immediate use from, while throwing in a good bit of more intermediate and advanced material to keep you learning for months, or even years, to come. We've tried to stay away from detailed "under the hood" explanations that, while certainly interesting, don't really help you learn or use PowerShell better. We've tried to keep examples short and easy to follow, so that you can use them as the basis for your learning.

And, speaking of examples, you'll find them all for download at www.SAPIENPress.com. Although we've made every effort to ensure the technical accuracy of these examples, changing products, typesetting errors, and other problems are sure to occur; by making the examples available for download, we can ensure that they're kept up-to-date and corrected. The Web site will also contain an "errata" list of any changes that we need to make to the text in order to keep it correct and accurate.

Note that the first edition of this text is based on the second Release Candidate (RC2) of Windows PowerShell; we don't anticipate any major corrections once PowerShell releases to manufacturing (RTM), but any corrections that do need to be made will be posted on the SAPIEN Press Web site for you, at no additional charge.

Thanks for choosing this book for your PowerShell education. Now, dive in: PowerShell awaits.

A handwritten signature in black ink that reads "Don Jones". The signature is written in a cursive, flowing style with a large initial "D" and "J".

| | |
|--|-------------------------------------|
| Foreword | iii |
| Part I: Introduction | 1 |
| Getting Started | 3 |
| What is PowerShell, and Why Should I Care? | 3 |
| PowerShell Requirements | 4 |
| Quick Start | 5 |
| Navigating Your System | 5 |
| Using the PowerShell Command Line | 7 |
| Aliases | 7 |
| Basic Cmdlets | 8 |
| Parameters | 9 |
| Ubiquitous Parameters | 9 |
| Snap-Ins..... | 9 |
| Profiles | 11 |
| Scripts | 13 |
| Redirection and Substitution..... | 14 |
| Variables | 15 |
| Variable Names and Intrinsic Variables | 16 |
| Variables are Objects..... | 16 |
| String Variables and Embedding | 17 |
| Parsing Mode..... | 18 |
| Special Characters | 19 |
| Scopes | 19 |
| Functions | 20 |
| Pipelines | 20 |
| Getting Help..... | 21 |
| Multiple Shells..... | 21 |
| Where Do You Go From Here? | 22 |
| How to Use This Book | 24 |
| Your First PowerShell Script | Error! Bookmark not defined. |
| Script Editing..... | Error! Bookmark not defined. |
| The Script..... | Error! Bookmark not defined. |
| Security Features | Error! Bookmark not defined. |
| Why Won't My Scripts Run? | Error! Bookmark not defined. |
| When Scripts Don't Run..... | Error! Bookmark not defined. |
| Digital Signatures..... | Error! Bookmark not defined. |
| Trusted Scripts..... | Error! Bookmark not defined. |

| | |
|--|-------------------------------------|
| Execution Policies | Error! Bookmark not defined. |
| Signing Scripts | Error! Bookmark not defined. |
| Alternate Credentials | Error! Bookmark not defined. |
| Is PowerShell Dangerous? | Error! Bookmark not defined. |
| Safer Scripts from the Internet | Error! Bookmark not defined. |
| Passwords and Secure Strings | Error! Bookmark not defined. |
| Technologies Overview | Error! Bookmark not defined. |
| Microsoft .NET Framework Essentials | Error! Bookmark not defined. |
| Reflection | Error! Bookmark not defined. |
| Assemblies | Error! Bookmark not defined. |
| Classes | Error! Bookmark not defined. |
| Advanced .NET in PowerShell | Error! Bookmark not defined. |
| Variables as Objects | Error! Bookmark not defined. |
| Variable Types | Error! Bookmark not defined. |
| Variable Precautions | Error! Bookmark not defined. |
| .NET Conclusion | Error! Bookmark not defined. |
| Understanding Scope | Error! Bookmark not defined. |
| Windows Management Instrumentation Essentials | Error! Bookmark not defined. |
| WMI Architecture | Error! Bookmark not defined. |
| WMI Documentation | Error! Bookmark not defined. |
| Working with Classes | Error! Bookmark not defined. |
| Remote Computers and WMI | Error! Bookmark not defined. |
| Configuring with WMI | Error! Bookmark not defined. |
| Part II: PowerShell Scripting | Error! Bookmark not defined. |
| Variables, Arrays, Objects and Escape Sequences | Error! Bookmark not defined. |
| Variables | Error! Bookmark not defined. |
| Get-Variable | Error! Bookmark not defined. |
| Environment Variables | Error! Bookmark not defined. |
| Set-Variable | Error! Bookmark not defined. |
| New-Variable | Error! Bookmark not defined. |
| Clear-Variable | Error! Bookmark not defined. |
| Remove-Variable | Error! Bookmark not defined. |
| Arrays | Error! Bookmark not defined. |
| Associative Arrays | Error! Bookmark not defined. |
| Creating an Associative Array | Error! Bookmark not defined. |
| Using an Associative Array | Error! Bookmark not defined. |
| Objects | Error! Bookmark not defined. |
| Properties | Error! Bookmark not defined. |
| Methods | Error! Bookmark not defined. |

| | |
|---|-------------------------------------|
| Variables as Objects..... | Error! Bookmark not defined. |
| Variable Types..... | Error! Bookmark not defined. |
| Variable Precautions..... | Error! Bookmark not defined. |
| Creating New Objects..... | Error! Bookmark not defined. |
| Escape Characters..... | Error! Bookmark not defined. |
| Operators | Error! Bookmark not defined. |
| Arithmetic Operators..... | Error! Bookmark not defined. |
| Precedence..... | Error! Bookmark not defined. |
| Variables..... | Error! Bookmark not defined. |
| Unary Operators..... | Error! Bookmark not defined. |
| Logical Operators..... | Error! Bookmark not defined. |
| Assignment Operators..... | Error! Bookmark not defined. |
| Bitwise Operators..... | Error! Bookmark not defined. |
| Special Operators..... | Error! Bookmark not defined. |
| Replace Operator..... | Error! Bookmark not defined. |
| Type..... | Error! Bookmark not defined. |
| Range Operator..... | Error! Bookmark not defined. |
| Call Operators..... | Error! Bookmark not defined. |
| Format Operator..... | Error! Bookmark not defined. |
| Comparison Operators..... | Error! Bookmark not defined. |
| Regular Expressions..... | Error! Bookmark not defined. |
| Introduction to Regular Expressions..... | Error! Bookmark not defined. |
| Writing Regular Expressions..... | Error! Bookmark not defined. |
| Regex Object..... | Error! Bookmark not defined. |
| Regular Expression Examples..... | Error! Bookmark not defined. |
| E-mail Address..... | Error! Bookmark not defined. |
| String with No Spaces..... | Error! Bookmark not defined. |
| Domain Credential..... | Error! Bookmark not defined. |
| Telephone Number..... | Error! Bookmark not defined. |
| IP Address..... | Error! Bookmark not defined. |
| Regular Expression Reference..... | Error! Bookmark not defined. |
| Loops and Decision Making Constructs | Error! Bookmark not defined. |
| ForEach..... | Error! Bookmark not defined. |
| For..... | Error! Bookmark not defined. |
| While..... | Error! Bookmark not defined. |
| Do While..... | Error! Bookmark not defined. |
| Do Until..... | Error! Bookmark not defined. |
| If..... | Error! Bookmark not defined. |
| Switch..... | Error! Bookmark not defined. |

| | |
|---|-------------------------------------|
| Break..... | Error! Bookmark not defined. |
| Continue..... | Error! Bookmark not defined. |
| Grouping, Sorting, Formatting, Exporting and More..... | 25 |
| Redirection..... | 25 |
| Out-File | 26 |
| Out-Printer | 27 |
| Tee-Object | 27 |
| Write-Host..... | 27 |
| Formatting..... | 29 |
| Format-List..... | 29 |
| Format-Table | 30 |
| Format-Wide | 32 |
| Format-Custom..... | 33 |
| GroupBy..... | 34 |
| Sort-Object..... | 35 |
| Where-Object..... | 38 |
| Exporting..... | 39 |
| Export-CSV | 39 |
| Export-CliXML..... | 41 |
| ConvertTo-HTML..... | 42 |
| System Forms..... | 44 |
| Modularization | Error! Bookmark not defined. |
| Script Blocks | Error! Bookmark not defined. |
| Functions | Error! Bookmark not defined. |
| Input Arguments..... | Error! Bookmark not defined. |
| Returning a Value | Error! Bookmark not defined. |
| Piping to Functions | Error! Bookmark not defined. |
| Function Phases | Error! Bookmark not defined. |
| Filters | Error! Bookmark not defined. |
| Functions vs. Filters | Error! Bookmark not defined. |
| Cmdlets and Snapins..... | Error! Bookmark not defined. |
| Modularization Tricks..... | Error! Bookmark not defined. |
| Debugging and Error Handling..... | Error! Bookmark not defined. |
| Handling Errors..... | Error! Bookmark not defined. |
| Error Actions | Error! Bookmark not defined. |
| Trapping Errors..... | Error! Bookmark not defined. |
| Trap Scope | Error! Bookmark not defined. |
| Throwing Your Own Exceptions..... | Error! Bookmark not defined. |
| Tips for Error Trapping..... | Error! Bookmark not defined. |

| | |
|---|-------------------------------------|
| Debugging Methodology..... | Error! Bookmark not defined. |
| Debug Mode and Tracing..... | Error! Bookmark not defined. |
| Managing Windows with PowerShell..... | Error! Bookmark not defined. |
| Managing Files and Directories..... | Error! Bookmark not defined. |
| Creating Text Files..... | Error! Bookmark not defined. |
| Reading Text Files..... | Error! Bookmark not defined. |
| Copying Files..... | Error! Bookmark not defined. |
| Deleting Files..... | Error! Bookmark not defined. |
| Renaming Files..... | Error! Bookmark not defined. |
| Creating Directories..... | Error! Bookmark not defined. |
| Listing Directories..... | Error! Bookmark not defined. |
| Deleting Directories..... | Error! Bookmark not defined. |
| Managing Systems with PowerShell and WMI..... | Error! Bookmark not defined. |
| Managing Services..... | Error! Bookmark not defined. |
| Listing Services..... | Error! Bookmark not defined. |
| Starting Services..... | Error! Bookmark not defined. |
| Stopping Services..... | Error! Bookmark not defined. |
| Managing Services..... | Error! Bookmark not defined. |
| Managing Permissions..... | Error! Bookmark not defined. |
| Managing Event Logs..... | Error! Bookmark not defined. |
| Managing Processes..... | Error! Bookmark not defined. |
| Managing the Registry..... | Error! Bookmark not defined. |
| Managing Directory Services..... | Error! Bookmark not defined. |
| Part III: Advanced PowerShell..... | Error! Bookmark not defined. |
| Shortcuts and Tips..... | Error! Bookmark not defined. |
| Command-Line Tips..... | Error! Bookmark not defined. |
| PrimalScript Tips..... | Error! Bookmark not defined. |
| Aliases and Shortcuts..... | Error! Bookmark not defined. |
| Memory Math..... | Error! Bookmark not defined. |
| Web-ify PowerShell..... | Error! Bookmark not defined. |
| Best Practices for PowerShell Scripting..... | Error! Bookmark not defined. |
| Best Practices for Variables..... | Error! Bookmark not defined. |
| Best Practices for Scripts..... | Error! Bookmark not defined. |
| Best Practice for Parameters..... | Error! Bookmark not defined. |
| Best Practices for Aliases..... | Error! Bookmark not defined. |
| Best Practices for Loops and Constructs..... | Error! Bookmark not defined. |
| Best Practices for Switches..... | Error! Bookmark not defined. |
| Best Practices for Comments..... | Error! Bookmark not defined. |
| Database Scripting..... | Error! Bookmark not defined. |

| | |
|---|-------------------------------------|
| Database Terminology..... | Error! Bookmark not defined. |
| Databases..... | Error! Bookmark not defined. |
| Tables..... | Error! Bookmark not defined. |
| Rows and Columns, Records and Fields..... | Error! Bookmark not defined. |
| Working with Databases..... | Error! Bookmark not defined. |
| Persisting DataTables..... | Error! Bookmark not defined. |
| Connecting to a Database..... | Error! Bookmark not defined. |
| Defining a Query..... | Error! Bookmark not defined. |
| Executing the Query..... | Error! Bookmark not defined. |
| Reading Data..... | Error! Bookmark not defined. |
| Finishing Up..... | Error! Bookmark not defined. |
| Database Example..... | Error! Bookmark not defined. |
| Extending Types..... | Error! Bookmark not defined. |
| The Types File..... | Error! Bookmark not defined. |
| Aliasing..... | Error! Bookmark not defined. |
| Adding Features..... | Error! Bookmark not defined. |
| Why is This Useful?..... | Error! Bookmark not defined. |
| PowerShell Jump-Start for VBScripters..... | Error! Bookmark not defined. |
| Variables..... | Error! Bookmark not defined. |
| COM Objects..... | Error! Bookmark not defined. |
| Instantiating Objects..... | Error! Bookmark not defined. |
| Using Objects..... | Error! Bookmark not defined. |
| GetObject..... | Error! Bookmark not defined. |
| Comments..... | Error! Bookmark not defined. |
| Loops and Constructs..... | Error! Bookmark not defined. |
| Type Conversion..... | Error! Bookmark not defined. |
| Operators and Special Values..... | Error! Bookmark not defined. |
| Functions and Subs..... | Error! Bookmark not defined. |
| Error Handling..... | Error! Bookmark not defined. |
| Windows Management Instrumentation..... | Error! Bookmark not defined. |
| Active Directory Services Interface..... | Error! Bookmark not defined. |
| Common Tasks in VBScript..... | Error! Bookmark not defined. |
| PowerShell Paradigm Change..... | Error! Bookmark not defined. |
| Cmdlet Reference..... | Error! Bookmark not defined. |
| Add-Content..... | Error! Bookmark not defined. |
| Add-History..... | Error! Bookmark not defined. |
| Add-Member..... | Error! Bookmark not defined. |
| Add-PSSnapin..... | Error! Bookmark not defined. |

| | |
|---------------------------------|-------------------------------------|
| Clear-Content | Error! Bookmark not defined. |
| Clear-Item | Error! Bookmark not defined. |
| Clear-ItemProperty | Error! Bookmark not defined. |
| Clear-Variable | Error! Bookmark not defined. |
| Compare-Object | Error! Bookmark not defined. |
| ConvertFrom-SecureString..... | Error! Bookmark not defined. |
| Convert-Path..... | Error! Bookmark not defined. |
| ConvertTo-Html | Error! Bookmark not defined. |
| ConvertTo-SecureString | Error! Bookmark not defined. |
| Copy-Item | Error! Bookmark not defined. |
| Copy-ItemProperty..... | Error! Bookmark not defined. |
| Export-Alias | Error! Bookmark not defined. |
| Export-Clixml | Error! Bookmark not defined. |
| Export-Console | Error! Bookmark not defined. |
| Export-Csv | Error! Bookmark not defined. |
| ForEach-Object..... | Error! Bookmark not defined. |
| Format-Custom..... | Error! Bookmark not defined. |
| Format-List..... | Error! Bookmark not defined. |
| Format-Table | Error! Bookmark not defined. |
| Format-Wide | Error! Bookmark not defined. |
| Get-Acl..... | Error! Bookmark not defined. |
| Get-Alias | Error! Bookmark not defined. |
| Get-AuthenticodeSignature | Error! Bookmark not defined. |
| Get-ChildItem..... | Error! Bookmark not defined. |
| Get-Command | Error! Bookmark not defined. |
| Get-Content | Error! Bookmark not defined. |
| Get-Credential | Error! Bookmark not defined. |
| Get-Culture | Error! Bookmark not defined. |
| Get-Date | Error! Bookmark not defined. |
| Get-EventLog..... | Error! Bookmark not defined. |
| Get-ExecutionPolicy | Error! Bookmark not defined. |
| Get-Help..... | Error! Bookmark not defined. |
| Get-History..... | Error! Bookmark not defined. |
| Get-Host..... | Error! Bookmark not defined. |
| Get-Item | Error! Bookmark not defined. |
| Get-ItemProperty | Error! Bookmark not defined. |
| Get-Location | Error! Bookmark not defined. |

| | |
|--------------------------|-------------------------------------|
| Get-Member..... | Error! Bookmark not defined. |
| Get-PfxCertificate | Error! Bookmark not defined. |
| Get-Process..... | Error! Bookmark not defined. |
| Get-PSDrive..... | Error! Bookmark not defined. |
| Get-PSPProvider..... | Error! Bookmark not defined. |
| Get-PSSnapin..... | Error! Bookmark not defined. |
| Get-Service..... | Error! Bookmark not defined. |
| Get-TraceSource | Error! Bookmark not defined. |
| Get-UICulture..... | Error! Bookmark not defined. |
| Get-Unique | Error! Bookmark not defined. |
| Get-Variable..... | Error! Bookmark not defined. |
| Get-WmiObject | Error! Bookmark not defined. |
| Group-Object | Error! Bookmark not defined. |
| Import-Alias..... | Error! Bookmark not defined. |
| Import-Clixml..... | Error! Bookmark not defined. |
| Import-Csv | Error! Bookmark not defined. |
| Invoke-Expression | Error! Bookmark not defined. |
| Invoke-History..... | Error! Bookmark not defined. |
| Invoke-Item | Error! Bookmark not defined. |
| Join-Path..... | Error! Bookmark not defined. |
| Measure-Command..... | Error! Bookmark not defined. |
| Measure-Object | Error! Bookmark not defined. |
| Move-Item..... | Error! Bookmark not defined. |
| Move-ItemProperty | Error! Bookmark not defined. |
| New-Alias..... | Error! Bookmark not defined. |
| New-Item..... | Error! Bookmark not defined. |
| New-ItemProperty..... | Error! Bookmark not defined. |
| New-Object | Error! Bookmark not defined. |
| New-PSDrive | Error! Bookmark not defined. |
| New-Service | Error! Bookmark not defined. |
| New-TimeSpan | Error! Bookmark not defined. |
| New-Variable | Error! Bookmark not defined. |
| Out-Default | Error! Bookmark not defined. |
| Out-File | Error! Bookmark not defined. |
| Out-Host..... | Error! Bookmark not defined. |
| Out-Null..... | Error! Bookmark not defined. |
| Out-Printer | Error! Bookmark not defined. |

| | |
|--------------------------------|-------------------------------------|
| Out-String..... | Error! Bookmark not defined. |
| Pop-Location..... | Error! Bookmark not defined. |
| Push-Location..... | Error! Bookmark not defined. |
| Read-Host..... | Error! Bookmark not defined. |
| Remove-Item..... | Error! Bookmark not defined. |
| Remove-ItemProperty..... | Error! Bookmark not defined. |
| Remove-PSDrive..... | Error! Bookmark not defined. |
| Remove-PSSnapin..... | Error! Bookmark not defined. |
| Remove-Variable..... | Error! Bookmark not defined. |
| Rename-Item..... | Error! Bookmark not defined. |
| Rename-ItemProperty..... | Error! Bookmark not defined. |
| Resolve-Path..... | Error! Bookmark not defined. |
| Restart-Service..... | Error! Bookmark not defined. |
| Resume-Service..... | Error! Bookmark not defined. |
| Select-Object..... | Error! Bookmark not defined. |
| Select-String..... | Error! Bookmark not defined. |
| Set-Acl..... | Error! Bookmark not defined. |
| Set-Alias..... | Error! Bookmark not defined. |
| Set-AuthenticodeSignature..... | Error! Bookmark not defined. |
| Set-Content..... | Error! Bookmark not defined. |
| Set-Date..... | Error! Bookmark not defined. |
| Set-ExecutionPolicy..... | Error! Bookmark not defined. |
| Set-Item..... | Error! Bookmark not defined. |
| Set-ItemProperty..... | Error! Bookmark not defined. |
| Set-Location..... | Error! Bookmark not defined. |
| Set-PSDebug..... | Error! Bookmark not defined. |
| Set-Service..... | Error! Bookmark not defined. |
| Set-TraceSource..... | Error! Bookmark not defined. |
| Set-Variable..... | Error! Bookmark not defined. |
| Sort-Object..... | Error! Bookmark not defined. |
| Split-Path..... | Error! Bookmark not defined. |
| Start-Service..... | Error! Bookmark not defined. |
| Start-Sleep..... | Error! Bookmark not defined. |
| Start-Transcript..... | Error! Bookmark not defined. |
| Stop-Process..... | Error! Bookmark not defined. |
| Stop-Service..... | Error! Bookmark not defined. |
| Stop-Transcript..... | Error! Bookmark not defined. |

| | |
|-----------------------------|-------------------------------------|
| Suspend-Service | Error! Bookmark not defined. |
| Tee-Object | Error! Bookmark not defined. |
| Test-Path | Error! Bookmark not defined. |
| Trace-Command | Error! Bookmark not defined. |
| Update-FormatData | Error! Bookmark not defined. |
| Update-TypeData | Error! Bookmark not defined. |
| Where-Object | Error! Bookmark not defined. |
| Write-Debug | Error! Bookmark not defined. |
| Write-Error | Error! Bookmark not defined. |
| Write-Host | Error! Bookmark not defined. |
| Write-Output | Error! Bookmark not defined. |
| Write-Progress | Error! Bookmark not defined. |
| Write-Verbose | Error! Bookmark not defined. |
| Write-Warning | Error! Bookmark not defined. |
| Type Reference | Error! Bookmark not defined. |
| Boolean | Error! Bookmark not defined. |
| DateTime | Error! Bookmark not defined. |
| Double | Error! Bookmark not defined. |
| Int | Error! Bookmark not defined. |
| String | Error! Bookmark not defined. |
| Advanced Types | Error! Bookmark not defined. |
| All Types | Error! Bookmark not defined. |
| Index | Error! Bookmark not defined. |

Part I: Introduction

PowerShell is not only a new scripting language, it's a brand-new way to administer Windows from a command-line interface. Learn what PowerShell is, quickly jump into PowerShell scripting, and review the technologies that lie underneath PowerShell's hood.

Windows PowerShell: TFM™
Sample Chapters

Chapter 1

Getting Started

In this chapter, we will take a brief tour of PowerShell and introduce you to some of its most important concepts. Many of these concepts will be covered in more detail in later chapters. For now, we want to help you become oriented with this new environment.

What is PowerShell, and Why Should I Care?

Administrators of UNIX and Linux systems (collectively referred to as “*nix” throughout this book) have always had the luxury of administrative scripting. In fact, most *nix operating systems are built on a command-line interface (CLI). While most *nix systems also feature a graphical user interface (GUI), the real work is done from the CLI. Every variant of *nix supports some sort of shell scripting language such as Bash that enables CLI commands to be strung together to automate administrative tasks.

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

In the late nineties Microsoft introduced Visual Basic Scripting Edition, which was commonly referred to as VBScript. This scripting language was compatible with Microsoft's Component Object Model (COM) that forms the building blocks of Windows itself. Because most GUI administrative tools were built on and utilized COM, it was felt that VBScript would provide a better automation environment. Unfortunately, VBScript

still only automated a fraction of Windows' capabilities. However, it can do far more than the simple CLI batch language that evolved from Microsoft's earliest MS-DOS operating system.

Both CLI tools and VBScript have other problems, the primary problem being consistency. Because both evolved over time, and were created by various groups within Microsoft who had no shared standards to work from, each CLI tool and COM interface (as used by VBScript) works a bit differently. This means every new tool or COM interface you use presents a new learning curve, which takes additional time that you may not have. All of this stems from the fact that Microsoft was never really committed to scripting and automation for Windows. In fact, the feeling was that it was *Windows*, which meant you primarily used the GUI to run it. However, as Windows' penetration into large companies and enterprises increased, managers and administrators accustomed to *nix began to demand the same scripting and automation capabilities from Windows. This brings us to *PowerShell*, which is Microsoft's first comprehensive, from-scratch effort to create a scriptable automation shell for Windows. PowerShell is built on the Microsoft .NET Framework, which has deep ties into almost every aspect of the operating system. Because Microsoft has made a strategic commitment to .NET, PowerShell's future is fairly secure since it will be built on the same platform on which most of Microsoft's other products will be built. In addition, above all else, PowerShell is *consistent*: There are clear guidelines for how PowerShell is to be built and extended, which means you won't have to learn an entirely new way of doing things every time you start a new script.

It's critical to recognize that PowerShell isn't a new scripting language a la VBScript. While PowerShell *has* a scripting language, it's actually an entirely new administrative interface. You can use it *interactively* without scripting, to issue commands to Windows and other Microsoft server products. You can also script it to automate more complex tasks. PowerShell is designed to be the place where an experienced administrator spends most of his or her time, replacing the GUI interfaces you've primarily used in the past in favor of a more productive, CLI-based administrative experience.

Exchange Server 2007 is perhaps the best example of how PowerShell can be leveraged. PowerShell was built into this version of Exchange from the outset. In fact, *all* of the product's administrative functionality was built in .NET and exposed through PowerShell, with the administrative GUI, or console, simply utilizing that underlying functionality. This means *any* Exchange administrative task can be performed in PowerShell, which, in turn, means *any* task can be scripted and automated in a consistent fashion. Whether or not the future use of PowerShell is equally comprehensive is up to the individual product groups within Microsoft. However, with Microsoft's strategic commitment both to .NET and administrative automation, it's probably a safe bet that PowerShell will finally offer a clear, consistent, and comprehensive tool for Windows administrative scripting.

PowerShell Requirements

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

certain situations. For example, PowerShell is part of the Exchange Server 2007 administrative tools.

Quick Start

PowerShell is easy to get up and running. First by simply running **PowerShell.EXE** (or select the Start Menu shortcut), you'll be in the new shell. PowerShell is a complete shell, which is not completely unlike the Cmd.exe shell with which you are probably already familiar. From within PowerShell, you can run normal GUI applications like Notepad or Calc, which open in their usual graphical windows. For applications that produce textual output (as opposed to using a GUI), you can capture the output within the PowerShell shell.

Under Cmd.exe, you typically ran CLI utilities like **Dir**, **Xcopy**, **Cacls**, and so forth. However, under PowerShell you'll primarily work with *cmdlets* (pronounced, "command-lets"). Cmdlets serve the same role in PowerShell as CLI tools did under Cmd.exe. The differences are they're all built to a consistent standard and they're all built using the .NET Framework. PowerShell scripting involves stringing these cmdlets together to perform various tasks. If you're a .NET developer, you can also write your own cmdlets.

Cmdlets are always named in a *verb-noun* format such as **Get-Process**. You can use the built-in **Get-Help** cmdlet to read help when available for other cmdlets. For example, **Get-Help Get-Process** displays help on the **Get-Process** cmdlet. However, as we'll cover later in this chapter, many cmdlets have *aliases* that make them work the same way as the familiar Cmd.exe commands. For example, the **Dir** command works fine under PowerShell because **Dir** is actually an alias to the appropriate cmdlet.

Navigating Your System

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

```
PS C:\>
```

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

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

Notice that drive names aren't limited to single letters. For example, the **HKLM** drive maps to the **HKEY_LOCAL_MACHINE** portion of the registry. Also notice the **Provider** column, which indicates the PowerShell provider or piece of software provides the connectivity to that particular resource. Providers are what make PowerShell so flexible. By simply adding a provider, you can gain access to entirely new resources through PowerShell. In addition to internal providers for aliases, functions, and variables, PowerShell ships with providers that give you access to certificates (the cert provider), the registry (HKCU and HKLM), the filesystem (drive letters), and the Windows environment (the env provider).

Additional Providers

Additional providers are available for Microsoft Visual SourceSafe and SharePoint services. Other providers will be available in the future, although these won't be covered or referenced in this book.

To change locations, simply use the **Set-Location** cmdlet, passing it the name of the location you want to change to as shown below.

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

Don't Worry About the Case

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

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

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

This will change the shell's focus directly to the indicated path. Note that the path is contained within double quotation marks because any argument that contains spaces *must* be enclosed within double quotation marks.

PowerShell also maintains a *stack* of locations. Think of the stack as a big pile of location names on top of one another. You can add or *push the current* location onto the top of the stack by using the **Push-Location** cmdlet. If you specify a path, you'll push the current location, and then change to the specified path. You can quickly change to the location on the top of the stack by using the **Pop-Location** cmdlet. For example, the following cmdlet moves the location C:\ to the top of the stack and changes to the C:\Test directory.

```
PS C:\>Push-Location C:\Test
```

Later, when you're ready to quickly change back to C:\, issue the following cmdlet:

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

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

Using the PowerShell Command Line

PowerShell has some very basic line-editing capabilities that you can use when typing at the command-line. These are not substitutes for a full development environment if you're writing scripts or cmdlets, but they provide basic features when you just need to run a script or cmdlet interactively such as:

- Down- and Up-Arrow displays previously entered commands.
- Left- and Right-Arrow move the cursor left and right, respectively.
- Home key moves to the beginning of the current command.
- End key moves to the end of the current command.
- Ctrl+Left and Ctrl+Right jump one word to the left and right.
- Insert toggles insert/overwrite mode.
- Backspace deletes the character in front of (to the left) your cursor.
- Delete removes the character to the right of your cursor.
- Press Tab to auto-complete path names and object members. This will be examined in more detail in later chapters.
- Esc (escape) clears the entire command-line so you can start over.

Aliases

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

```
PS C:\>Set-Alias popl Pop-Location
```

Now you can use **Popl** in place of **Pop-Location**. You can remove, or undefine, an alias by using the generic **Remove-Item** cmdlet:

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

This removes the **Popd** alias from the system. PowerShell predefines a number of useful aliases. You can see a list of these aliases by running **Get-Alias**. Here's a portion of the output you'll see:

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

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

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

Aliases can *only* be for a single cmdlet or external executable, not for any accompanying parameters or arguments. However, you *can* write a script that performs more complex tasks such as executing cmdlets that have arguments, and create an alias to the script. When you define an alias, it only lasts for the current PowerShell session. To make an alias permanent, add it in your profile, which is a special PowerShell script that runs when you start the shell. We'll talk more about profiles in a bit.

Basic Cmdlets

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

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

In Chapter 9, we'll talk you through output formatting, which helps prevent information from getting lost "off the edge" of PowerShell's 80-column display window.

Using **Get-Command** is a good way to inventory what your capabilities within PowerShell are. For any given cmdlet, **Get-Command** tells you more about it. For example, the following example describes what the **Set-Alias** cmdlet does:

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

As shown below, you can also use wildcards to get **Get-Help** to learn more specific information:

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

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

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

Parameters

Many cmdlets can accept parameters that are passed by name using a hyphen, then the parameter (or argument) name followed by a space, and then the value you're passing to the parameter. The following example creates a new file.

```
PS C:\>New-Item -type file "myfile.txt"
```

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

Ubiquitous Parameters

Most cmdlets support a set of *ubiquitous parameters*, which are always optional. This means ubiquitous parameters such as those listed below don't need to be specified if you don't want to use them.

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

Snap-Ins

A *snap-in* is essentially a collection of cmdlets. For example, running the **Get-Pssnapin** cmdlet shows the snap-ins provided with the basic shell as shown below:

```
PS C:\> get-pssnapin
```

```
Name       : Microsoft.PowerShell.Core
PSVersion  : 1.0
Description : This PSSnapIn contains MSH management cmdlets used to man
            onents affecting the MSH engine.

Name       : Microsoft.PowerShell.Host
PSVersion  : 1.0
Description : This PSSnapIn contains cmdlets used by the MSH host.

Name       : Microsoft.PowerShell.Management
PSVersion  : 1.0
Description : This PSSnapIn contains general management Cmdlets used to
            Windows components.

Name       : Microsoft.PowerShell.Security
PSVersion  : 1.0
Description : This PSSnapIn contains cmdlets to manage MSH security.

Name       : Microsoft.PowerShell.Utility
PSVersion  : 1.0
Description : This PSSnapIn contains utility cmdlets used to manipulate
```

You can add the **-pssnapin** parameter to the **Get-Command** cmdlet to see which cmdlets are in a particular snap-in. Below is a partial list for the **Microsoft.PowerShell.Utility** snap-in:

```
PS C:\> get-command -pssnapin microsoft.powershell.utility
```

| CommandType | Name | Definition |
|-------------|-----------------|----------------|
| ----- | ---- | ----- |
| Cmdlet | Add-Member | Add-Member [-M |
| Cmdlet | Clear-Variable | Clear-Variable |
| Cmdlet | Compare-Object | Compare-Object |
| Cmdlet | ConvertTo-Html | ConvertTo-Html |
| Cmdlet | Export-Alias | Export-Alias [|
| Cmdlet | Export-Clixml | Export-Clixml |
| Cmdlet | Export-Csv | Export-Csv [-P |
| Cmdlet | Format-Custom | Format-Custom |
| Cmdlet | Format-List | Format-List [[|
| Cmdlet | Format-Table | Format-Table [|
| Cmdlet | Format-Wide | Format-Wide [[|
| Cmdlet | Get-Alias | Get-Alias [[-N |
| Cmdlet | Get-Culture | Get-Culture [- |
| Cmdlet | Get-Date | Get-Date [[-Da |
| Cmdlet | Get-Host | Get-Host [-Ver |
| Cmdlet | Get-Member | Get-Member [[- |
| Cmdlet | Get-TraceSource | Get-TraceSourc |

As you'll learn later in this chapter, you can make custom versions of PowerShell that include just the snap-ins you want, which essentially creates task-specific versions of the shell.

But wait a moment. What will happen if you create a script that relies on a particular snap-in being present, and share that script with an administrator who doesn't *have* that snap-in? The answer is that the script won't work. One way you can help remember which snap-ins a script requires, is to use a self-documenting feature in PowerShell. For example, you can add something like the following to your script:

```
#requires -MshSnapIn Microsoft.PowerShell.Utility
```

Of course, you don't need to specify this for the core snap-ins because they're always present, so there's no way to create a shell without them. However, if you're using nonstandard snap-ins, it's a good idea to add `#requires` to your script.

Profiles

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

- `%windir%\system32\WindowsPowerShell\v1.0\profile.ps1`
This profile is loaded for all users.
- `%windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1`
This profile is loaded for all users, and only for the default instance of PowerShell.
- `%UserProfile%\My Documents\WindowsPowerShell\profile.ps1`
This profile is loaded per-user, and affects all versions of PowerShell which are installed.
- `%UserProfile%\My Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1`
This profile is loaded per-user, but only affects the default instance of PowerShell.

Note that not all of these profiles exist by default, but you can create any ones that you need to use. If these files exist, they are read in this order - conflicts are “won” by whichever file is read last. If none of these files exist, PowerShell will use its built-in default settings. So what's in a profile? Listed below is an example profile that installs with PowerShell.

```
# Copyright (c) 2005 Microsoft Corporation. All rights reserved.
#
# THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY
# OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED
# TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
# PARTICULAR PURPOSE

set-alias cat          get-content
set-alias cd           set-location
set-alias clear        clear-host
set-alias cp           copy-item
set-alias h            get-history
set-alias history      get-history
set-alias kill         stop-process
set-alias lp           out-printer
set-alias ls           get-childitem
set-alias mount        new-drive
set-alias mv           move-item
set-alias popd         pop-location
set-alias ps           get-process
set-alias pushd        push-location
set-alias pwd          get-location
set-alias r            invoke-history
set-alias rm           remove-item
set-alias rmdir        remove-item
set-alias echo         write-object

set-alias cls          clear-host
set-alias chdir        set-location
set-alias copy         copy-item
set-alias del          remove-item
set-alias dir          get-childitem
set-alias erase        remove-item
set-alias move         move-item
set-alias rd           remove-item
set-alias ren          rename-item
set-alias set          set-variable
set-alias type         get-content

function help
{
    get-help $args[0] | out-host -paging
}

function man
{
    get-help $args[0] | out-host -paging
}

function mkdir
{
    new-item -type directory -path $args
}

function md
{
    new-item -type directory -path $args
}

function prompt
{
    "PS " + $(get-location) + "> "
}

& {
    for ($i = 0; $i -lt 26; $i++)
    {
        $funcname = ([System.Char]($i+65)) + ':'
        $str = "function global:$funcname { set-location $funcname } "
        invoke-command $str
    }
}
```

```
}  
}
```

This profile is really little more than a “startup script.” It starts by setting several cmdlet aliases so that familiar commands like **Cd** (the MS-DOS “change directory” command) are aliased to an equivalent PowerShell cmdlet (**Set-Location** in this case). It then declares several functions. For example, the **Get-Help** cmdlet exists to access PowerShell’s built-in help files. The **Help** function defined by this profile calls **Get-Help**, passes the name of the help file you asked for (that’s the \$args[0] part), and then pipes the output to the **Out-Host** cmdlet that has the -paging parameter. This means your help will appear on one screen at a time instead of scrolling by too quickly to read. The coolest part of this sample profile is the last bit, which takes some careful decoding:

```
& {  
    for ($i = 0; $i -lt 26; $i++)  
    {  
        $funcname = ([System.Char]($i+65)) + ':'  
        $str = "function global:$funcname { set-location $funcname } "  
        invoke-command $str  
    }  
}
```

The **&** operator means *invoke* or, “Please run the following code.” So everything within the outer set of curly braces will execute. In other words, it’s a code block that will run each time you start PowerShell with this profile. So what does it do?

A **for** loop (which we’ll cover in Chapter 8) is set to count from zero to 25. That is, the **for** loop starts at zero, continues while it’s less than 26, and increments by 1 each time through the loop. Within the loop a variable named \$funcname is created. The loop’s current value (0 to 25) is added to 65, and an ASCII character of that value is selected. For example, 65 is the letter “A” and 66 is the letter “B”. A colon character is appended, making the value of \$funcname something like “A:” or “B:” all the way to “Z:”. All of these look like drive letters, right?

A variable named \$str is then created. The first time through the loop, it contains the string “function global:A: {set-location A: }”. You can see where “A:” (and then “B:”, then “C:”, and so forth) are inserted at the location of \$funcname. The **Invoke-Command** cmdlet then executes whatever code is inside \$str. Essentially, this loop is declaring 26 new functions named A:, B:, etc. that execute the **Set-Location** cmdlet. What’s the point of all this? In the old Cmd.exe shell, you could type a drive letter like C: to “change” to that drive. Since by default PowerShell doesn’t support that functionality, these 26 dynamically-declared functions provide that capability, making PowerShell work a bit more like the old, familiar Cmd.exe shell. This is the power of a profile - Setting up PowerShell to work the way you want.

All of these aliases and functions become available right within the PowerShell environment. You can modify this example to create your own profile or create your own profile entirely from scratch. Simply use the same script editor or development environment such as SAPIEN PrimalScript that you use to write PowerShell scripts.

Scripts

So far, we looked at ways to run cmdlets directly and view their output. PowerShell scripts are designed to string several cmdlets together to automate more complex tasks. PowerShell scripts must have the filename extension .PS1; to run a script, simply type its

name without the extension. PowerShell will look in the environment Path variable or any files with the .PS1 extension to find your script. If your script takes input arguments, then type them after the script name:

```
. Myscript arg1 arg2
```

Did you notice the period at the beginning of the line? This period tells PowerShell to execute the script in the current *scope*, which we'll discuss in a bit. Scripts use their own language, which is similar to both VBScript and C#. Most of the remainder of this book will be spent discussing this scripting language.

How Can it Tell?

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

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

Redirection and Substitution

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

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

This cmdlet gives you a file named Commandref.txt that contains the output of the **Get-Command** cmdlet. This file is located in the current location, as indicated by the PowerShell prompt.

Append Output

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

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

```
PS C:\>Get-ChildItem $(Read-Host -Prompt "Enter file path/name: ")
```

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

```
PS C:\> Get-ChildItem $(Read-Host -Prompt "Enter file path/name: ")  
Enter filename: : C:\
```

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

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

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

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

```
PS C:\>New-Item -type file $(1..5)
```

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

Variables

Like many scripting environments, PowerShell supports the creating and use of *variables*. Think of a variable as a container that has a name and can hold values. For example, the following command creates a new variable named \$var and assigns it the numeric value 100.

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

Keep in mind that variable names always begin with \$ in PowerShell.

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

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

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

Variables can also contain the output of cmdlets. For example, the following runs the **Get-Process** cmdlet and put its entire output into the variable \$a:

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

Did you notice that `$a` was never *declared*, as some scripting languages require or allow you to do? PowerShell doesn't need variables to be declared in advance, which means you can make up and use new variables as you go. In fact, unlike some languages, PowerShell is designed so, as a general rule, variables *aren't* declared in advance.

Variable Names and Intrinsic Variables

Variable names can contain *any* character. However, as shown below, the variable must be enclosed in curly braces if it doesn't start with a letter.

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

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

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

This variable name writes “Hello!” to a text file named `C:\File.txt`. Remember that *every resource* PowerShell connects to can be presented with a file-like path. For example, the path `HKLM\SOFTWARE` goes to the registry. This can be a powerful technique for quickly changing values in various resources.

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

Variables are Objects

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

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

This variable assigns the value “Hello, World” to the variable `$var`. `$var` now contains that variable, but `$var` also has a number of capabilities as an object (something we'll touch in more in Chapters 4 and 5). One of the capabilities is the **SubString()** method, which is listed below.

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

This calls the **Write-Host** cmdlet, which outputs text to the console. It asks that the `$var` object execute its **SubString()** method, which starts at the third character position (numbering begins with 0, so 2 is the third character) and takes 2 characters. This will output “ll” to the console. Similarly, the following example outputs the number 12, because that's how long the contents of `$var` are: 12 characters

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

Note that PowerShell doesn't visually differentiate between variables that contain strings and those that contain numbers:

```
$var = 5  
$var = "Hello"
```

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

```
PS C:\> $var = 3  
PS C:\> write-host $var.length
```

```
PS C:\> $var = "Hello"  
PS C:\> write-host $var.length  
5
```

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

So, you're probably wondering where we found out about **Length**, **SubString**, and so forth. The short answer is - Chapter 5, where we'll cover variables in more depth.

However, a longer answer is that PowerShell can sort of *tell* you. Type **\$a = "hello"** into PowerShell and hit Enter. Then type **\$a.** (be sure to include a period after the letter "a") and hit Tab. PowerShell will start to list each *member* of the String object of which your variable \$a is an instance. A new member will be listed each time you hit Tab.

Eventually, you'll come to **SubString**. This use of Tab is a type of text-based substitute for the code completion features (which go by brand names such as IntelliSense and PrimalSense) in graphical development environments.

String Variables and Embedding

String variables treat embedded variables in an interesting fashion. For example, consider the following where \$var2 is embedded in a literal string:

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

In this example, the value "Hello" was assigned to \$var. The value "\$var, World!" was assigned to \$var2. When passed to **Write-Host**, \$var was expanded, which means its *contents* were displayed. This occurs because \$var2 was assigned using double quotation marks. Now, consider a similar example:

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

Notice the difference? This time, the value passed into \$var2 was contained in *single quotation marks* instead of double, which prevented \$var from being expanded. So the literal value "\$var, World!" was stored in \$var2, as evidence by the **Write-Host** output.

\$var2 is still considered a string object, and either single quotes or double quotes can be used to contain strings.

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

For example:

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

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

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

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

Parsing Mode

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

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

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

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

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

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

Rather than having to type:

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

This would be cumbersome and unintuitive, since it's not the way past Windows shells have worked.

This is all called the shell *parsing mode*, whether the shell treats things as strings by default or not. The rules are pretty simple:

- If the first character is a number, a variable (\$), or a quoted string, then the shell works in *expression mode*, in which all strings must be quoted.

- If the first character is a letter, ampersand (&), or a dot followed by a space or a letter, then the shell works in *command mode*, which is where everything is assumed to be a string unless it's a variable or is in parentheses, as we've demonstrated.

Special Characters

Sometimes, you may need to display special characters that can't be typed. For this reason, PowerShell provides an *escape character* of ```, which is a backward apostrophe, which is technically called a *grave accent mark*. This escape character is usually located on the same key as `~` on your keyboard. To display a ``` by itself, type ````. The special characters are:

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

Scopes

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

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

This example shows why it was so important to specify the period - to ensure that the script would run in the *current* scope, thus having access to any variables or functions already declared within that scope. If we hadn't used the period, PowerShell would take its default action of creating a new scope for the script. This technique of preceding the script name with a period and a space is called *dot sourcing*, which essentially makes the script behave as if you are typing each line of the script directly into the PowerShell shell. Unless you use dot sourcing, by default all scripts are run in a newly created scope. *Child scopes*, or scopes created by another scope, can *read* variables from the parent scope, but cannot *change* them as easily. *Parent scopes* cannot access child scope variables in any way.

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

Scope Names

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

Here's an example of scopes. Say a script declares a variable named `$var`. A function runs, and also declares `$var`, which means there are two copies of `$var` in existence: one in the script's scope and one in the function's scope. Because the function is contained within the script, its scope is a child of the script's scope. That means the function *can* access script-level variables by referring to `$script:var` if it chooses to do so. That is, the function can access the name of the scope (script) and the name of the variable from that scope (var). More information on functions is coming up next.

Variables can also be declared as *private*, which means they're accessible *only* from the current scope, and not from within child scopes. The following example declares a private variable named var:

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

Here's another example. Suppose you create a variable in the basic shell, without running a script. That variable exists in the shell's global scope. If you then run a script, that script gets its own scope, which is a child of the shell itself. Therefore, by default, the script has read access to the variable you declared in the shell because the script's scope is a child of the shell's scope.

We're going to cover scopes in a lot more detail in Chapter 4 since they're somewhat confusing at first, and since they can dramatically affect the way your scripts run.

Functions

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

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

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

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

Pipelines

One of the most powerful and possibly confusing aspects of PowerShell is its *data pipelines*, which provide a means of passing data and objects from one cmdlet to another in a very robust fashion. Perhaps you've used the `Cmd.exe More` utility such as the one shown below to slow down the display of a long directory.

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

This takes the output of **Dir** and “pipes” it to **More**, which displays the data in nice pages, and waits for you to hit a key before displaying the next page.

Pipes have the same basic function in PowerShell. In fact, that character in between **Dir** and **More** is called the *pipe character*, which is usually on the backslash key of your keyboard. Here’s a robust example:

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

PowerShell also has a **More** command if you want to pipe multi-screen output to it for one-page-at-a-time display.

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

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

```
ProcessName : csrss
Id           : 1080

ProcessName : explorer
Id           : 1952

ProcessName : Groove
Id           : 2656

ProcessName : inetinfo
Id           : 1524
```

(This output is truncated for illustrative purposes.) The ability of pipes to pass data to other cmdlets, and the ability of PowerShell to deal with complex, structured objects in a text interface such as the Process object, is part of what has people so excited about PowerShell. You can, in just a single line of code, complete fairly complex administrative tasks.

Getting Help

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

Multiple Shells

So far, we’ve been working with the “default” PowerShell shell that cannot be modified that includes the PowerShell binaries and cmdlets. However, PowerShell understands the concept of *multiple* available shells. For example, Exchange Server 2007 includes a different shell than the default that includes the various cmdlets associated directly with

Exchange Server (all of those cmdlets are bundled into Exchange-specific snap-ins). You can build a new shell that includes the PowerShell binaries and snap-ins, as well as any additional snap-ins you obtain or create. For example, this allows you to create a shell that's specific for a particular job task in your environment, ensuring that all related cmdlets stay packaged together. It also helps protect the integrity of the original default shell so you don't mess it up by accident.

The **Make-Shell** cmdlet lets you create a new shell. It manages the compilation process and takes care of all the messy details so shell creation is relatively simple. It only takes two parameters: **-out**, which specifies where the new shell will go, and **-namespace**, which names the *runspace* configuration. The runspace configuration is one of the parts of the shell that is automatically generated, and contains the list of cmdlets, providers, and other parts the shell needs to run. An optional parameter is **-reference**, which allows you to add cmdlets to the shell. For example:

```
PS> make-shell -out MyNewShell -namespace MyShellWithMyCmdlet  
-reference mycmdlet.dll
```

You can launch your new shell from right within PowerShell as follows:

```
./MyNewShell
```

The **Make-Shell** cmdlet actually has a lot of other optional parameters that allow you to specify a custom program icon, build scripts into the shell, etc. You can run **Help Make-Shell** to obtain a complete list of these optional parameters. Of course, making a new shell isn't the *only* way to add cmdlets or providers. In Chapter 11 you'll learn about snap-ins that are a way of adding cmdlets and providers do the current session of the shell.

Where Do You Go From Here?

This has been a whirlwind tour of PowerShell's core features and capabilities to whet your appetite. So what's next?

- In Chapter 2, you'll get a chance to write a useful PowerShell script so you can quickly see this new shell in action.
- Chapter 3 focuses on PowerShell security, which is very important to securing and protecting both your scripts and your environment.
- An overview of PowerShell technologies in Chapter 4 will help you understand what PowerShell is built on, and provide a background for core technologies that you'll utilize within PowerShell.

Then we'll move into the core of PowerShell scripting in the next group of chapters:

- Chapter 5 begins a look at how PowerShell deals with data including variables, arrays, and objects. These are really the building blocks of PowerShell scripting.
- Chapter 6 covers operators, which is how values are compared and what PowerShell's logical constructs are built upon.
- Chapter 7 covers regular expressions, which is a powerful technique for comparison string values to one another.
- Chapter 8 covers those logical constructs including loops and decision-making constructs that give your scripts some intelligence.

- Creating attractive output with PowerShell is easy, and Chapter 9 shows you how to do it in a variety of ways.

We're on to advanced topics in the last few chapters:

- Modularization is the topic of Chapter 10, where you'll learn to create and use functions, script blocks, cmdlets, and snapins.
- Error handling and debugging—two things you'll definitely want to learn—are covered in Chapter 11.
- Finally, Chapter 12 wraps up with a supply of real-world examples of PowerShell scripting including scripts to work with WMI, services, Access Control Lists (ACLs), event logs, processes, the registry, directory services, and the Web.

Chapter 13 through 16 are relatively short, and cover some highly-advanced topics that not everyone will need—but we wanted to include them for folks who do.

- Chapter 13 is a small collection of tips and tricks we've learned while using PowerShell.
- Chapter 14 covers database scripting, including a short tutorial on the basic concepts and some examples of using databases from within PowerShell.
- PowerShell's internal data types can be extended, and Chapter 15 shows you how.
- Chapter 16 is for readers who already know VBScript. While there's no easy conversion of scripts to PowerShell, we wanted to use this chapter to give you a jump start by relating your VBScript skills to similar things in PowerShell.

Finally, two appendices provide a basic cmdlet reference and a reference to PowerShell's major data types.

How to Use This Book

We want to make sure you have everything you need to effectively use this book and PowerShell. However, you may notice that this book is lacking one important thing that you may have expected it to contain - a CD full of sample scripts.

The sample scripts can be found online at no charge at www.SAPIENPress.com. We included all the scripts online to make sure we can keep them updated and corrected at all times. For your convenience, scripts can be downloaded individually or in a single ZIP file.

SAPIEN Press' Web site also contains detailed errata for the book including corrections and amplifications, along with discussion forums related to the book's contents.

For more PowerShell scripting assistance, visit www.ScriptingAnswers.com, where you'll find free discussion forums about PowerShell, PowerShell tutorials, a ScriptVault full of VBScript and PowerShell scripts, and other free resources.

In this chapter, we examined at how PowerShell works and touched on most of its major features including variables and pipelines. Remember that later chapters will explore almost all of these topics in more detail. However, at this point you should be comfortable running PowerShell and using it interactively to perform very basic tasks such as running commands.

Chapter 9

Grouping, Sorting, Formatting, Exporting and More

Although PowerShell is a console-based management shell, because it is object-oriented there are many different ways it can present data. In this chapter we'll look at the many ways you can manipulate, format, and even export the output from your PowerShell commands and scripts.

Because PowerShell's output is almost entirely text-based, it's easy to mistake it for a text-based shell. However, what is really happening behind the scenes is that PowerShell is using .NET and cmdlets to carry out your commands and manipulate data as needed. Only when all processing is complete is the final data formatted for textual presentation. However there are many things you can do to control how the data is ultimately presented.

Redirection

Many times you may want to save the results of a cmdlet to a text file. In PowerShell this is easily accomplished with redirection using `>` or `>>`. In Chapter 1 we saw that `>` redirects console output to a file and overwrites any existing file, while `>>` appends to an existing file. If you use `>>` and the file doesn't already exist, then the file will be created:

```
PS C:\> get-wmiobject -class win32_computersystem >audit.txt
PS C:\> get-wmiobject -class win32_operatingsystem >>audit.txt
PS C:\> get-content audit.txt
```

```
Domain           : SAPIENPRESS
Manufacturer     : Dell Computer Corporation
Model            : Latitude D800
Name             : GODOT
PrimaryOwnerName : Administrator
TotalPhysicalMemory : 1609805824
```

```
SystemDirectory : C:\WINDOWS\system32
Organization     : TestCo
BuildNumber      : 2600
RegisteredUser   : Administrator
SerialNumber     : 55274-640-1614466-00000
Version          : 5.1.2600
```

```
PS C:\>
```

In this example we've sent the output of the **Get-Wmiobject** cmdlets to a text file called `audit.txt`. The first command creates the file and the second appends to the file. We'll examine other ways to control what type of output PowerShell produces, all of which you can save to a text file using console redirection.

Out-File

PowerShell includes a cmdlet that sends output to a text file. When saving output to a file, this cmdlet is easier to use in a script instead of trying to use redirection. You will use this cmdlet in a pipeline:

```
PS C:\> Get-service |out-file c:\logs\audit.txt
```

If you open `c:\logs\audit.txt`, you'll see the results of the **Get-Service** cmdlet. The cmdlet also makes it easy to append to an existing file:

```
PS C:\> Get-process |out-file c:\logs\audit.txt -append
```

Unlike redirection, you can also instruct **Out-File** to not overwrite an existing file:

```
PS C:\> get-service |out-file c:\logs\audit.txt -noclobber
Out-File : File C:\logs\audit.txt already exists and NoClobber was
```

```
specified.
```

You can see that PowerShell refused to overwrite an existing file. However, it's is easy enough to force PowerShell overwrite an existing file:

```
PS C:\> get-wmiobject -class win32_logicaldisk |out-file  
c:\logs\audit.txt -force  
PS C:\> get-content c:\logs\audit.txt
```

```
DeviceID      : C:  
DriveType     : 3  
ProviderName  :  
FreeSpace     : 2815569920  
Size          : 15726702592  
VolumeName    : Server2003  
PS C:\>
```

Here we sent the output of the **Get-Wmiobject** cmdlet to the same file and forced it to overwrite the existing file.

Out-Printer

One of the many new PowerShell features is the ability to send a cmdlet's output directly to a printer using the **Out-Printer** cmdlet:

```
PS C:\> get-wmiobject -class win32_logicaldisk |out-printer
```

This command sends the output of the **Get-Wmiobject** cmdlet directly to the default printer. If you have more than one printer, you can specify a printer by name:

```
PS C:\> get-wmiobject -class win32_logicaldisk |out-printer "Adobe  
PDF"
```

In this instance we're sending the output the virtual printer that is installed with Adobe Acrobat. This results in a new pdf with the output of the **Get-Wmiobject** cmdlet. If you want to send output to a network printer, simply specify the printer UNC:

```
PS C:\> get-process |out-printer "\\Print01\HPLaserJ"
```

Tee-Object

As we've seen so far with the redirection examples, if we send the output to a file, we don't see the output at the console. What if we want to do both? Using PowerShell's **Tee-Object** cmdlet we can view the output and send it to a text file:

```
PS C:\>Get-process |tee-object c:\processes.txt
```

This expression displays the results of the **Get-Process** cmdlet and sends them to the text file. Unfortunately, the **Tee-Object** doesn't support sending output to a printer.

Write-Host

You've seen this cmdlet used many times in this book. It is used to provide information to you while a script or command block is executing. The information is outside of the command pipeline. In other words, the cmdlet does not affect the objects that are being manipulated. Instead, it provides the feedback you requested:

```
PS C:\> foreach ($svc in (get-service)) {  
>> if ($svc.status -eq "running") {  
>> write-host "Running: " $svc.DisplayName  
>> }  
>> }  
>>
```

```
Running: Windows Audio  
Running: AVG7 Alert Manager Server  
Running: AVG7 Update Service  
Running: AVG E-mail Scanner  
Running: Background Intelligent Transfer  
Running: Cryptographic Services  
Running: DCOM Server Process Launcher  
Running: DHCP Client  
Running: DNS Client  
Running: Event Log  
Running: COM+ Event System  
Running: IIS Admin  
...
```

In this example, the status of each service is checked. If it is running, then we call **Write-Host** to help display a message.

One of the slickest features with **Write-Host** is the ability to colorize the output. **Write-Host** has two optional parameters: **-backgroundcolor SystemColor** and **-foregroundcolor SystemColor**. You can use either or both of these parameters. In addition, you can select any system color from this list:

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

Here's an example of how you might use this feature:

ServiceDemo.ps1

```
#ServiceDemo.ps1

$services=Get-wmiobject -class "Win32_service"
foreach ($svc in $services) {
  if (($svc.startmode -eq "Auto") -AND ($svc.state -ne "Running")) {
    write-host $svc.displayname $svc.state $svc.startmode `
      -backgroundcolor "White" -foregroundcolor "Red"
  }
  else
  {
    write-host $svc.displayname $svc.state $svc.startmode
  }
}$services=Get-wmiobject -class "Win32_service"
foreach ($svc in $services) {
  if (($svc.startmode -eq "Auto") -AND ($svc.state -ne "Running")) {
    write-host $svc.displayname "["$svc.state"]" "["$svc.startmode"]" `
      -backgroundcolor "White" -foregroundcolor "Red"
  }
  else
  {
    write-host $svc.displayname "["$svc.state"]" "["$svc.startmode"]"
  }
}
```

This script examines each service as queried from WMI, and displays the service's display name, status, and start mode. However, we've added one PowerShell feature. Any service with a start type of "Auto" but is not running will be displayed in a red font with a white background to make it stand out when the script is run.

Formatting

Just about every PowerShell cmdlet is designed to produce textual output. The cmdlet developer creates a default output format based on the information to be delivered. For example, the output of **Get-Process** is a horizontal table. However, if you need a different output format, PowerShell has a few choices that are discussed below.

Format-List

This cmdlet produces a columnar list. Here's a sample using **Get-Process**:

```
PS C:\> get-process | format-list
Id      : 720
Handles : 63
CPU     : 0.1301872
Name    : ApntEx

Id      : 584
Handles : 105
CPU     : 0.5107344
Name    : Apoint

Id      : 404
Handles : 130
CPU     : 0.4706768
Name    : avgamsvr

Id      : 444
Handles : 205
CPU     : 2.1130384
Name    : avgcc
...
```

Even though we've truncated the output, you get the idea. Instead of the regular horizontal table, each process and its properties is listed in a column. As we've pointed when this cmdlet has been used in other examples, the **Format-List** doesn't use all the properties that you get with the regular cmdlet output. PowerShell tries to help by presenting the information you are likely to need in this format. If you prefer more control over what information is displayed, you can use the **-property** cmdlet parameter to specify the properties:

```
PS C:\> get-process winword |format-list -property `
>> name,workingset,id,path
>>
```

```
Name      : WINWORD
WorkingSet : 32522240
Id        : 564
Path      : C:\Program Files\Microsoft Office\OFFICE11\WINWORD.EXE
```

```
PS C:\>
```

In this example we've called **Get-Process** seeking specific information on the WinWord process.

How Did You Know?

You might wonder how we knew what properties can be displayed when we use the **-property** cmdlet. To review, it is important to get to know the **Get-Member** cmdlet. This command lists all the available properties for the process object:

```
get-process | get-member
```

Different cmdlets and objects have different properties, especially in WMI.

Format-Table

Just as there are some cmdlets that use a table format as the default, there are some that use a list format. Of course, sometimes you may prefer a table. The format of this **Get-Wmiobject** expression produces a list by default:

```
PS C:\> get-wmiobject -class win32_logicaldisk
```

```
DeviceID      : C:  
DriveType    : 3  
ProviderName :  
FreeSpace    : 2815565824  
Size         : 15726702592  
VolumeName   : Server2003
```

```
DeviceID      : D:  
DriveType    : 5  
ProviderName :  
FreeSpace    :  
Size         :  
VolumeName   :
```

```
DeviceID      : E:  
DriveType    : 3  
ProviderName :  
FreeSpace    : 2891620352  
Size         : 24280993792  
VolumeName   : XP
```

This is not too hard to read. However, here's the same cmdlet except using the **Format-Table**:

```
PS C:\> get-wmiobject -class win32_logicaldisk |format-table
```

| DeviceID | DriveType | ProviderName | FreeSpace | Size | VolumeName |
|----------|-----------|--------------|------------|-------------|------------|
| C: | 3 | | 2815565824 | 15726702592 | Server2003 |
| D: | 5 | | | | |
| E: | 3 | | 2891620352 | 24280993792 | XP |

```
PS C:\>
```

Since the ProviderName property is blank, we can clean-up this output even more by using **-property** as we did with **Format-List**:

```
PS C:\> get-wmiobject -class win32_logicaldisk |format-table `  
>> -property deviceID,freespace,size,volumename,drivetype  
>>
```

| deviceID | freespace | size | volumename | drivetype |
|----------|------------|-------------|------------|-----------|
| C: | 2815565824 | 15726702592 | Server2003 | 3 |
| D: | | | | 5 |
| E: | 2891239424 | 24280993792 | XP | 3 |

```
PS C:\>
```

Notice that the property headings are in the same order that we specified in the expression. They also use the same case.

This cmdlet lets you tweak the output by using **-autosize**, which automatically adjusts the table output based on the date:

```
PS C:\> get-wmiobject -class win32_logicaldisk |format-table `
>> -property deviceID,freespace,size,volumename,drivetype -autosize
>>
```

| deviceID | freespace | size | volumename | drivetype |
|----------|------------|-------------|------------|-----------|
| C: | 2815565824 | 15726702592 | Server2003 | 3 |
| D: | | | | 5 |
| E: | 2890489856 | 24280993792 | XP | 3 |

```
PS C:\>
```

This is the same command as before, except it includes `autosize`. Notice how neater the output is. Using **-autosize** eliminates the need to calculate how long lines will be, add padding or scripting voodoo. Now you can format output to meet your requirements without all the string manipulation when using the traditional `Cmd.exe` shell or even VBScript.

Format-Wide

Some cmdlets, like **Get-Service**, produce a long list of information that scrolls off the console screen. Wouldn't it be nice to get this information in multiple columns across the console screen? We can accomplish this with the **Format-Wide** cmdlet:

```
PS C:\> get-service |format-wide
```

| | |
|------------------------|--------------------------------|
| Alerter | ALG |
| AppMgmt | aspnet_state |
| AudioSrv | Avg7Alrt |
| Avg7UpdSvc | AVGEMS |
| BAsfIpM | BITS |
| Browser | CiSvc |
| ClipSrv | clr_optimization_v2.0.50727_32 |
| COMSysApp | CryptSvc |
| CVPND | DcomLaunch |
| Dhcp | dmadmin |
| dmserver | Dnscache |
| ERSvc | Eventlog |
| EventSystem | FastUserSwitchingCompatibility |
| GrooveAuditService | GrooveInstallerService |
| GrooveRunOnceInstaller | helpsvc |
| HidServ | HTTPFilter |
| ... | |

If you prefer more than two columns, which is the default, use the `-column` parameter to specify the number of columns:

```
PS C:\> get-service |format-wide -column 3
```

```
Alerter                ALG                    AppMgmt
aspnet_state           AudioSrv              Avg7Alrt
Avg7UpdSvc             AVGEMS               BasfIpM
BITS                  Browser              CiSvc
ClipSrv                clr_optimization_v2.0.5... COMSysApp
CryptSvc              CVPND                DcomLaunch
Dhcp                  dmadmin              dmserver
Dnscache              ERSvc                Eventlog
...
```

However, don't get carried away. The more columns you specify, the more you'll find the output getting truncated.

The **Get-Service** cmdlet also let's you specify which single property you would like to display:

```
PS C:\> get-service |format-wide displayname -column 3
```

```
Alerter                Application Layer Gatew... Application Man...
ASP.NET State Service  Windows Audio          AVG7 Alert Mana...
AVG7 Update Service   AVG E-mail Scanner     Broadcom ASF IP...
Background Intelligent . Computer Browser      Indexing Service
ClipBook              .NET Runtime Optimizati... COM+ System App...
Cryptographic Services Cisco Systems, Inc. VPN... DCOM Server Pro...
DHCP Client           Logical Disk Manager Ad... Logical Disk Ma...
DNS Client            Error Reporting Service Event Log
COM+ Event System     Fast User Switching Com... Groove Audit Serv
Groove Installer Service GrooveRunOnceInstaller Help and Support
...
```

Unlike **Format-Table** and **Format-List** that allow multiple properties, **Format-Wide** only permits a single property. In this example we've specified the service's display name.

Format-Custom

PowerShell provides the ability for you to customize how data is presented. Unfortunately it requires defining a new format in a custom XML file, then using the **Update-FormatData** cmdlet to register it in PowerShell. Frankly, for most administrators this cmdlet requires more effort than it's worth since it requires a certain degree of knowledge about .NET classes.

With this in mind, PowerShell has one custom format that lists the object class and properties that can be useful. Here's an example using the **Get-Process** cmdlet:

```
PS C:\> get-process winword |format-custom
```

```
class Process
{
  Id = 2308
  Handles = 431
  CPU = 31.0546544
  Name = WINWORD
}
```

```
PS C:\>
```

Compare this output to the same expression, but each using a different format:

```
PS C:\> get-process winword|format-table
```

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id | ProcessName |
|---------|--------|-------|-------|-------|--------|------|-------------|
| ----- | ----- | ----- | ----- | ----- | ----- | -- | ----- |
| 470 | 19 | 37076 | 60908 | 207 | 72.60 | 2308 | WINWORD |

```
PS C:\> get-process winword|format-list
```

Id : 2308
Handles : 470
CPU : 72.6044
Name : WINWORD

```
PS C:\> get-process winword|format-wide
```

WINWORD

```
PS C:\>
```

Of course it's up to you to decide which format meets your needs for a given task.

From the Architect

Jeffrey Snover, the PowerShell architect, has a helpful blog entry titled "Use of Wildcards in PowerShell Formatting" that discusses customizing output. You should be able to find this blog entry at:

<http://blogs.msdn.com/powershell/archive/2006/04/29/586775.aspx>

GroupBy

All the format cmdlets include a parameter called **-GroupBy** that allows you to group output based on a specified property. For example, here is a **Get-Service** expression that groups services by their status such as Running or Stopped. The output below has been edited for brevity.

```
PS C:\> get-service |format-table -groupby status

        Status: Stopped

Status   Name                DisplayName
-----   -
Stopped  Alerter              Alerter
Stopped  ALG                  Application Layer Gateway Service
Stopped  AppMgmt              Application Management
Stopped  aspnet_state         ASP.NET State Service

        Status: Running

Status   Name                DisplayName
-----   -
Running  AudioSrv             Windows Audio
Running  Avg7Alrt             AVG7 Alert Manager Server
Running  Avg7UpdSvc           AVG7 Update Service
Running  AVGEMS               AVG E-mail Scanner

        Status: Stopped

Status   Name                DisplayName
-----   -
Stopped  BAsfIpM              Broadcom ASF IP monitoring service ...
Stopped  BITS                 Background Intelligent Transfer Ser...
Stopped  Browser              Computer Browser
Stopped  CiSvc                Indexing Service
Stopped  ClipSrv              ClipBook
Stopped  clr_optimizatio...   .NET Runtime Optimization Service v...
Stopped  COMSysApp            COM+ System Application

        Status: Running

Status   Name                DisplayName
-----   -
Running  CryptSvc             Cryptographic Services

        Status: Running

Status   Name                DisplayName
-----   -
Running  wuau servicing       Automatic Updates
Running  WZCSVC               Wireless Zero Configuration

        Status: Stopped

Status   Name                DisplayName
-----   -
Stopped  xmlprov              Network Provisioning Service

PS C:\>
```

As you can see, grouping helps a little bit. However, this is probably not what you expected since the cmdlet appears to be first grouping services alphabetically and then grouping them by status.

Sort-Object

What we are really after is sorting the output first and then grouping it. The **Sort-Object** cmdlet does exactly what its name implies - it sorts objects based on property values.

```
PS C:\> get-process | sort-object handles
```

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id | ProcessName |
|---------|--------|-------|-------|-------|--------|------|-------------|
| 0 | 0 | 0 | 16 | 0 | | 0 | Idle |
| 21 | 1 | 168 | 376 | 4 | 0.46 | 776 | smss |
| 30 | 2 | 1912 | 2436 | 29 | 0.11 | 1288 | cmd |
| 34 | 2 | 400 | 1524 | 15 | 0.21 | 2664 | WLTRYSVC |
| 43 | 2 | 376 | 1392 | 13 | 0.18 | 2932 | MsPMSPSV |
| 62 | 3 | 1820 | 4404 | 34 | 0.21 | 1456 | ApntEx |
| 64 | 3 | 1744 | 5628 | 37 | 0.24 | 324 | notepad |
| 65 | 2 | 1472 | 1600 | 14 | 0.17 | 2488 | wdfmgr |
| 69 | 3 | 632 | 2044 | 13 | 0.10 | 1580 | sqlbrowser |
| 72 | 3 | 828 | 2428 | 27 | 0.33 | 1208 | scardsvr |
| 76 | 2 | 524 | 2116 | 19 | 0.04 | 828 | avgupsvc |
| 91 | 5 | 1284 | 3184 | 29 | 2.16 | 300 | svchost |
| 95 | 4 | 1528 | 5852 | 39 | 0.83 | 844 | sqlmangr |

...

Here we've taken the output of **Get-Process** and sorted it by the handles property. The default sort is ascending, but if you prefer the cmdlet includes a **-descending** parameter:

```
PS C:\> get-process | sort-object handles -descending
```

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id | ProcessName |
|---------|--------|-------|-------|-------|--------|------|-------------|
| 1290 | 49 | 14000 | 23608 | 127 | 38.52 | 1892 | svchost |
| 1076 | 0 | 0 | 220 | 2 | 47.18 | 4 | System |
| 817 | 78 | 15856 | 8436 | 136 | 15.55 | 1628 | Groove |
| 706 | 7 | 1888 | 4880 | 28 | 18.51 | 868 | csrss |
| 616 | 16 | 25680 | 41096 | 127 | 100.01 | 484 | explorer |
| 589 | 12 | 8076 | 1508 | 62 | 3.24 | 892 | winlogon |
| 538 | 11 | 15508 | 8864 | 95 | 126.88 | 1948 | Smc |
| 483 | 20 | 38092 | 63020 | 223 | 168.63 | 2308 | WINWORD |

...

Let's return to our earlier example in which we tried to group the output of **Get-Service** by status. Now we can pipe the **Get-Service** cmdlet to **Sort-Object**, specifying primary sort on status, then on name. Next we send the object to **Format-Table** and group by status. Here's the output we get:

```
PS C:\> get-service|sort-object status,name |format-table `
>>-groupby status
>>

    Status: Stopped

Status   Name                DisplayName
-----   -
Stopped  Alerter             Alerter
Stopped  ALG                 Application Layer Gateway Service
Stopped  AppMgmt            Application Management
Stopped  aspnet_state       ASP.NET State Service
Stopped  BAsfIpM           Broadcom ASF IP monitoring service ...
Stopped  BITS              Background Intelligent Transfer Ser...
Stopped  Browser           Computer Browser
Stopped  CiSvc             Indexing Service
...

    Status: Running

Status   Name                DisplayName
-----   -
Running  AudioSrv          Windows Audio
Running  Avg7Alrt          AVG7 Alert Manager Server
Running  Avg7UpdSvc       AVG7 Update Service
Running  AVGEMS           AVG E-mail Scanner
Running  CryptSvc         Cryptographic Services
Running  DcomLaunch       DCOM Server Process Launcher
Running  Dhcp             DHCP Client
Running  Dnscache         DNS Client
Running  Eventlog         Event Log
Running  EventSystem      COM+ Event System
Running  IISADMIN         IIS Admin
Running  lanmanserver     Server
Running  lanmanworkstation Workstation
...

PS C:\>
```

Again, we've edited the output for brevity, but you get the picture. One final **Sort-Object** parameter is **-Unique**, which not only gives sorted output, but it also displays only the unique values:

```
PS C:\> $var=@(7,3,4,4,4,2,5,5,4,8,43,54)
PS C:\> $var|sort
2
3
4
4
4
4
5
5
7
8
43
54
PS C:\> $var|sort -unique
2
3
4
5
7
8
43
54
PS C:\>
```

We've defined an array of numbers and first pipe it through a regular **Sort-Object** cmdlet. Compare that to the second expression that uses **-Unique**. Now the output is sorted and only unique objects are returned.

Alias Alert

You will probably find it easier to use the alias for Sort-Object, which is Sort, as we did in the last example.

PowerShell also has a **Get-Unique** cmdlet that functions essentially the same as Sort - **Unique**, but without the sorting feature. Here's the array we just used piped through **Get-Unique**:

```
PS C:\> $var|get-unique
7
3
4
2
5
4
8
43
54
PS C:\>
```

Where-Object

In addition to sorting, you may need to limit or filter the output. The **Where-Object** cmdlet is a filter that lets you control what data is ultimately displayed. This cmdlet is almost always used in a pipeline expression where output from one cmdlet is piped to this cmdlet. The **Where-Object** cmdlet requires a code block enclosed in braces that is executed as the filter.

Here's an expression to get all instances of the Win32_Service class where the state property of each object equals stopped.

```
get-wmiobject -class win32_service | where {$_.state -eq "Stopped"}
```

You may want to further refine this expression and format the output by piping to yet another cmdlet:

```
PS C:\> get-wmiobject -class win32_service | `
>>where {$_.state -eq "Stopped"} | format-wide
>>
```

```
Alerter
AppMgmt
BasfIpM
CiSvc
clr_optimization_v2.0.50727_32
CVPND
dmserver
FastUserSwitchingCompatibility
GrooveInstallerService
helpsvc
NetDDE
Netlogon
NtmsSvc
PDEngine
PolicyAgent
RDSessMgr
rpcapd
RSVP
SQLAgent$CRM
SQLWriter
SwPrv
TlntSvr
upnphost
VMAuthdService
vmount2
VSS
WmdmPmSN
wscsvc
ALG
aspnet_state
Browser
ClipSrv
COMSysApp
dmadmin
ERSvc
GrooveAuditService
GrooveRunOnceInstaller
HidServ
NetDDEdsdm
NtLmSsp
ose
Pml Driver HPZ12
RasAuto
RemoteAccess
RpcLocator
SharedAccess
SQLAgent$MICROSOFTSMLBIZ
SSDPSRV
SysmonLog
TrkWks
UPS
VMnetDHCP
VMware NAT Service
W3SVC
Wmi
xmlprov
```

```
PS C:\>
```

In this example we've taken the same **Get-Wmiobject** expression and piped it through **Format-Wide** to get a nice two column report.

The key is recognizing that the script block in braces is what filters the object. If nothing matches, the filter then nothing will be displayed.

Exporting

PowerShell's ability to manipulate objects is pretty formidable. We've seen how PowerShell permits you to control the output format of an expression or cmdlet. However, PowerShell even has the ability to change or export the object into something else.

Export-CSV

A comma separated value (CSV) file is a mainstay of administrative scripting. It's a text-based database that can be parsed into an array or opened in a spreadsheet program like Microsoft Excel. The cmdlet requires an input object that is typically the result of a piped cmdlet:

```
Get-process | export-csv processes.csv
```

When you run this command on your system it creates a text file called processes.csv. When the file is opened in a spreadsheet program, you'll be amazed by the amount of information that is available. In fact, it's probably overkill for most situations.

Here's another version of basically the same expression except this time we're using **Select-Object** to specify the properties we want returned:

```
PS C:\> get-process |select-object name,id,workingset,cpu | `
>> export-csv processes.csv
>>
```

```
PS C:\> get-content processes.csv
#TYPE System.Management.Automation.PSCustomObject
Name,Id,WorkingSet,CPU
acrotray,3996,6574080,0.7110224
ApntEx,1456,4718592,6.5894752
Apoint,1592,7147520,6.0787408
avgamsvr,436,7389184,4.155976
avgcc,1684,12288000,10.4550336
avgemc,860,22593536,9.5036656
avgupsvc,828,3182592,1.6824192
BCMWLTRY,2948,6508544,16.2333424
Client,1084,1019904,140.6121904
cmd,1288,1093632,0.5207488
csrss,868,3440640,82.7790304
cypnd,4000,8015872,9.6538816
EXCEL,2452,6545408,6.6996336
explorer,484,35528704,801.6427056
firefox,3028,71385088,881.2872288
Groove,2032,11628544,25.3264176
Idle,0,16384,
inetinfo,3012,5357568,1.4420736
Microsoft.Crm.Application.Host,1796,28168192,14.7812544
MSASCui,1732,10907648,7.310512
MsMpEng,1832,14114816,107.8951456
MsPMSPSv,2932,1425408,1.0114544
nsvsc32,1788,3522560,2.5236288
powershell,1560,53522432,7.9314048
procexp,588,10452992,86.1438688
rapimg,2380,6225920,3.4850112
PS C:\>
```

This produces a raw data report that we can further process any way we want. For example, if the **Out-File** already exists it will be overwritten unless you use **-NoClobber**. If you don't want the #TYPE header, which we find distracting, specify **-NoTypeInformation** as part of the **Export-CSV** cmdlet.

On a related note, PowerShell also has an **Import-CSV** cmdlet that reads the contents of the csv file and displays the data as a table. Here's an example with abbreviated output:

```
PS C:\> import-csv processes.csv
```

| Name | Id | WorkingSet | CPU |
|----------|------|------------|------------|
| ---- | -- | ----- | --- |
| acrotray | 3996 | 6574080 | 0.7110224 |
| ApntEx | 1456 | 4718592 | 6.5894752 |
| Apoint | 1592 | 7147520 | 6.0787408 |
| avgamsvr | 436 | 7389184 | 4.155976 |
| avgcc | 1684 | 12288000 | 10.4550336 |
| avgemc | 860 | 22593536 | 9.5036656 |
| avgupsvc | 828 | 3182592 | 1.6824192 |
| cmd | 1288 | 1093632 | 0.5207488 |
| csrss | 868 | 3440640 | 82.7790304 |
| cvpnd | 4000 | 8015872 | 9.6538816 |
| EXCEL | 2452 | 6545408 | 6.6996336 |
| ... | | | |

Export-CliXML

If you prefer to store results as an XML file, perhaps for processing by other tools, you can use PowerShell's **Export-CliXML** cmdlet. It works much the same way as **Export-CSV**:

```
PS C:\> get-wmiobject -class win32_processor | export-clixml wmiproc.xml
```

This creates an XML file called `wmiproc.xml` that can be imported back into PowerShell using **Import-CliXML**:

```
PS C:\> import-clipxml wmiproc.xml

AddressWidth           : 32
Architecture           : 0
Availability            : 3
Caption                : x86 Family 6 Model 9 Stepping 5
ConfigManagerErrorCode : 
ConfigManagerUserConfig : 
CpuStatus              : 1
CreationClassName      : Win32_Processor
CurrentClockSpeed      : 1598
CurrentVoltage         : 33
DataWidth              : 32
Description            : x86 Family 6 Model 9 Stepping 5
DeviceID               : CPU0
ErrorCleared           : 
ErrorDescription       : 
ExtClock               : 133
Family                 : 2
InstallDate            : 
L2CacheSize            : 1024
L2CacheSpeed           : 
LastErrorCode          : 
Level                  : 6
LoadPercentage         : 
Manufacturer           : GenuineIntel
MaxClockSpeed          : 1598
Name                   : Intel(R) Pentium(R) M processor 1600MHz
OtherFamilyDescription : 
PNPDeviceID            : 
PowerManagementCapabilities : 
PowerManagementSupported : False
ProcessorId            : A7E9F9BF00000695
ProcessorType          : 3
Revision               : 2309
Role                   : CPU
SocketDesignation      : Microprocessor
Status                 : OK
StatusInfo             : 3
Stepping               : 5
SystemCreationClassName : Win32_ComputerSystem
SystemName             : GODOT
UniqueId               : 
UpgradeMethod          : 6
Version                : Model 9, Stepping 5
VoltageCaps            : 2
__GENUS                : 2
__CLASS                : Win32_Processor
...
PS C:\>
```

As with the other exporting cmdlets, you can use **-NoClobber** to avoid overwriting an existing file.

ConvertTo-HTML

Finally, PowerShell includes a cmdlet to convert text output to an HTML table with the **ConvertTo-HTML** cmdlet. At its simplest, you can run an expression like this:

```
PS C:\> Get-Service | ConvertTo-HTML
```

If you execute this expression you'll see HTML code fly across the console, which doesn't do you much good. This can be changed by piping the HTML output to a file using **Out-File**, specifying a file name:

```
PS C:\> Get-Service | ConvertTo-HTML | out-file services.html
```

Now when you open services.html in a Web browser, you'll see a pretty complete table of running services and their properties. By default, the cmdlet lists all properties.

However, you can specify the properties by name and in whatever order you prefer:

```
PS C:\> Get-Service | ConvertTo-HTML Name,DisplayName,Status | `
>> out-file services.html
>>
```

```
PS C:\>
```

Now when you open services.html it's a little easier to work with. If you want to dress-up the page a bit, **ConvertTo-HTML** has some additional parameters as shown in the following table:

Table: ConvertTo-HTML Optional Parameters

| Parameter | Description |
|-----------|---|
| Head | Inserts text into the <head> tag of the html page. You might want to include metadata or style sheet references. |
| Title | Inserts text into the <title> tag of the html page. This let's you have a more meaningful title to the page other than the default HTMLTABLE. |
| Body | Inserts text within the <body></body> tag. This lets you specify body specify formatting like fonts and colors as well as any text you want to appear before the table. |

Here's a script where we put it all together.

Service2HTML.ps1

```
#Service2HTML.ps1
# a style sheet, style.css, should be in the same directory
# as the saved html file.

$server=hostname
$body="Services Report for "+$server.ToUpper()+"<HR>"
$file="c:\\"+$server+"-services.html"

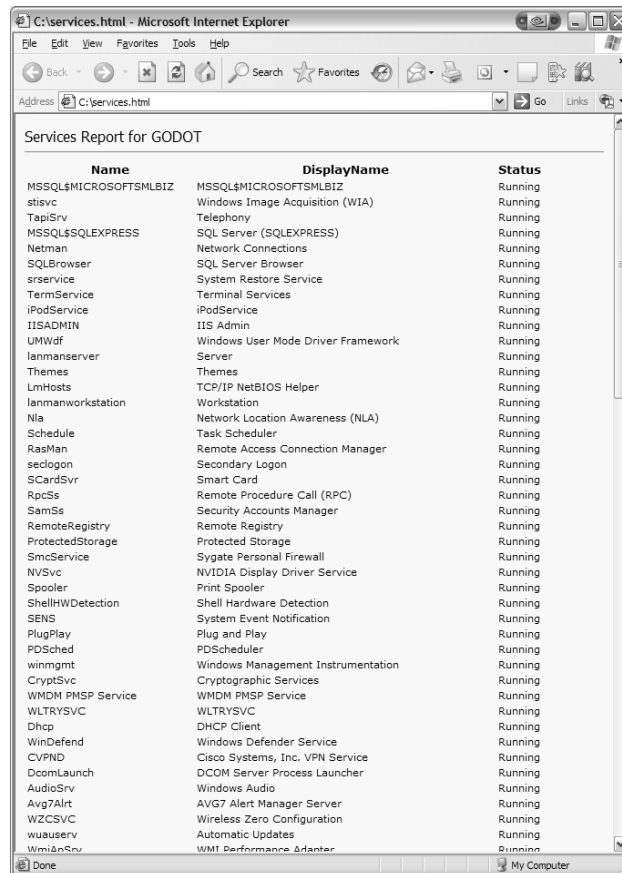
write-host "Generating Services Report for "$server.ToUpper()

get-service |sort -property status -descending | ConvertTo-HTML `
Name,DisplayName,Status -Title "Service Report" `
-Head "<link rel=stylesheet type=text/css href=style.css>" `
-Body $body | out-file $file

write-host "Report Generation Complete! Open" $file "for results."
```

This script takes the **Get-Service** cmdlet and generates a formatted HTML page. The script starts by defining some variables. First we want the computer name to use in the report and other variables. Then we define a variable for the **-Body** parameter. If just text is being used, we don't have to bother with this. However, the **ConvertTo-HTML** cmdlet is a little finicky and doesn't handle the results of embedded cmdlets very well.

By defining a variable we can ensure its value is a string. We also specify the location and name of the saved file. We're using the server name as part of the filename. After a message is sent to the user informing him the report is being generated, the heart of the script is reached. We take the **Get-Service** cmdlet and first pipe it to the **Sort-Object** cmdlet, sorting on service status and returning the results in descending order. This puts Running services at the top of the page and Stopped services at the bottom. Next we pipe that to **ConvertTo-HTML** specifying the properties we want in the table. We include a **-head** parameter so we can reference a style sheet and then the **-body** parameter using the \$body variable we defined at the beginning of the script. All of this is piped to **Out-File**, which saves the result to an HTML file. The results can be seen in Figure 9-1.



| Name | DisplayName | Status |
|------------------------|------------------------------------|---------|
| MSSQL\$MICROSOFTSMLBIZ | MSSQL\$MICROSOFTSMLBIZ | Running |
| stisvc | Windows Image Acquisition (WIA) | Running |
| TapiSrv | Telephony | Running |
| MSSQL\$SQLEXPRESS | SQL Server (SQLEXPRESS) | Running |
| Netman | Network Connections | Running |
| SQLBrowser | SQL Server Browser | Running |
| srsservice | System Restore Service | Running |
| TermService | Terminal Services | Running |
| iPodService | iPodService | Running |
| IISADMIN | IIS Admin | Running |
| UMWdf | Windows User Mode Driver Framework | Running |
| lanmanserver | Server | Running |
| Themes | Themes | Running |
| LmHosts | TCP/IP NetBIOS Helper | Running |
| lanmanworkstation | Workstation | Running |
| Nla | Network Location Awareness (NLA) | Running |
| Schedule | Task Scheduler | Running |
| RasMan | Remote Access Connection Manager | Running |
| seclogon | Secondary Logon | Running |
| SCardSvr | Smart Card | Running |
| RpcSs | Remote Procedure Call (RPC) | Running |
| SamSs | Security Accounts Manager | Running |
| RemoteRegistry | Remote Registry | Running |
| ProtectedStorage | Protected Storage | Running |
| SmcService | Sygate Personal Firewall | Running |
| NVSvc | NVIDIA Display Driver Service | Running |
| Spooler | Print Spooler | Running |
| ShellHWDetection | Shell Hardware Detection | Running |
| SENS | System Event Notification | Running |
| PlugPlay | Plug and Play | Running |
| PDSched | PDScheduler | Running |
| winnmgt | Windows Management Instrumentation | Running |
| CryptSvc | Cryptographic Services | Running |
| WMDM PMSP Service | WMDM PMSP Service | Running |
| WLTRYSVC | WLTRYSVC | Running |
| Dhcp | DHCP Client | Running |
| WinDefend | Windows Defender Service | Running |
| CVPND | Cisco Systems, Inc. VPN Service | Running |
| DcomLaunch | DCOM Server Process Launcher | Running |
| AudioSrv | Windows Audio | Running |
| Avg7Alht | AVG7 Alert Manager Server | Running |
| WZCSVC | Wireless Zero Configuration | Running |
| wuauerv | Automatic Updates | Running |
| Wmi | WMI Performance Adapter | Running |

Figure 9-1 Service2HTML.ps1 saved HTML page

System Forms

Even though PowerShell is a console-based shell, it is built on .NET. This means we have access to many of the .NET objects. For example, we can use .NET forms to create graphical interfaces for PowerShell scripts.

Advanced Stuff

Working with the Windows forms is beyond basic PowerShell. However, we want to give you a taste of the power behind PowerShell. Documentation for the System.Windows.Forms namespace can be found at: [http://msdn2.microsoft.com/en-us/library/k50ex0x9\(vs.80\).aspx](http://msdn2.microsoft.com/en-us/library/k50ex0x9(vs.80).aspx).

First let's look at a brief example in this script:

```
DemoMsgBox.ps1
#DemoMsgBox.ps1
[void][Reflection.Assembly]::LoadWithPartialName("`
System.Windows.Forms")
$MsgBox = [Windows.Forms.MessageBox]
$button=[Windows.Forms.MessageBoxButtons]::OK
$icon=[windows.forms.MessageBoxIcon]::Information
$MsgBox::show("Hello world", "Demo Msg Box", $button, $icon)
```

In order to work with system forms, we need to load the .NET assembly, which is done in the first line. The use of [void] prevents any information about the assembly from being displayed by PowerShell. At this point normally we would call **New-Object** to create the message box object, but PowerShell sometimes doesn't know everything. If you try to use **New-Object**, PowerShell will complain that it doesn't have a constructor for Windows.Forms.MessageBox or a System.Windows.Forms.MessageBox. It doesn't matter that this is a perfectly valid .NET object. However, we can manipulate the assembly directly to define a button and icon, and then display the message box shown in Figure 9-2.



Figure 9-2 PowerShell Message Box

To be honest, this is too much work. If you want to display a message box, it's a good idea to leverage the Windows Script Host object:

DemoPopup.ps1

```
#DemoPopup.ps1
$Shell=new-object -COM wscript.shell
$msg="Hello World"
$buttons=0+64
$shell.popup($msg,5,"Demo Popup",$buttons)
```

This script assumes you have some experience with VBScript. If this is true, you will recognize the Wscript.Shell popup that is essentially a message box with a timer. In this example, the popup will display for five seconds with the Information icon.

VBScript Alert

If you use this technique, you can't use references like VBOKOnly and VBExclamation. Instead, you need to use the actual values such as 0 and 64. You values can be found in the VBScript documentation.

If you want more control over your forms, you can use a script like this:

HelloForm.ps1

```
#HelloForm.ps1
[void][Reflection.Assembly]::LoadWithPartialName(`
"System.Windows.Forms")
$Form = New-Object System.Windows.Forms.Form
#default form size is 300x300 pixels
$Form.width=250
$form.height=200
$Label=new-object System.Windows.Forms.Label
$Label.Text="Hello World"
$Label.visible=$true
$Form.Text = "PowerShell TFM"
$Button = New-Object System.Windows.Forms.Button
$Button.Text = "OK"
#set button vertical button position
$Button.Top=$Form.Height*.50

#default button width is 75
#Center button horizontally
$Button.left=($Form.Width*.50)-75/2

$Button.Add_Click({$Form.Close()})
$Form.Controls.Add($Button)
$Form.Controls.Add($Label)
$Form.ShowDialog()
```

As you can see, you have to define everything when working with System forms. In this instance, PowerShell understands how to create a System.Windows.Forms.Form object, which makes our work a little easier. We define the form size, then add and define a text label and button. Be sure to add the controls to the form, otherwise you'll never see them. By the way, we've added `_Click` method for the button, which has the form close itself. Finally, we call the `ShowDialog()` method to display the form shown in Figure 9-3.



Figure 9-3: PowerShell Form

We'll wrap up this chapter with a script inspired by a blog posting on Abhishek's PowerShell Blog (<http://spaces.msn.com/abhishek225/>) that uses a DataGrid form to display information.

ServicesGrid.ps1

```
#ServicesGrid.ps1

[void][reflection.assembly]::LoadWithPartialName(`
"System.Windows.Forms")
[void][reflection.assembly]::LoadWithPartialName("System.Drawing")

$form = new-object System.Windows.Forms.Form
$form.Size = new-object System.Drawing.Size 400,500
$form.Text = "PowerShell TFM"

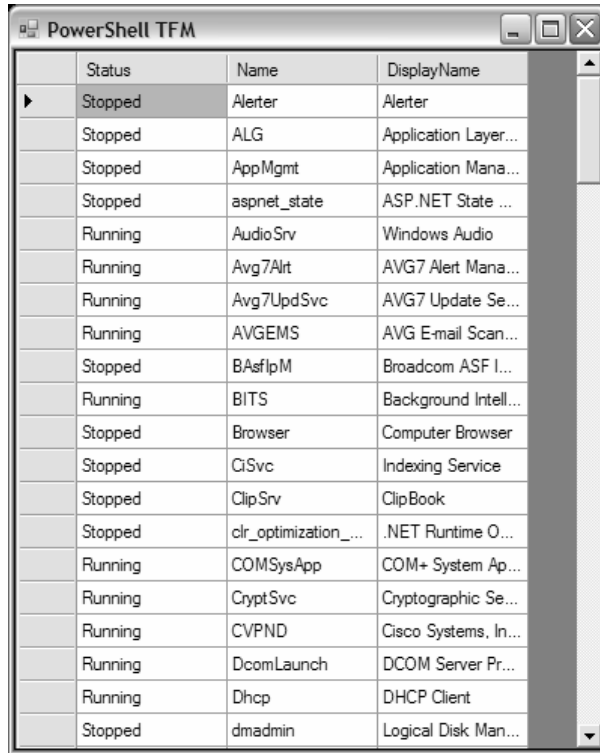
$dataGridView = new-object System.Windows.Forms.DataGridView

$array= new-object System.Collections.ArrayList

$data=@(get-service | write-output)
$array.AddRange($data)
$dataGridView.DataSource = $array
$dataGridView.Dock = [System.Windows.Forms.DockStyle]::Fill
$dataGridView.AllowUserToResizeColumns=$True

$form.Controls.Add($dataGridView)
$form.topmost = $True
$form.ShowDialog()
```

We won't go into detail on how the script works. In general you can see where new objects are created and properties are defined. After creating the form and datagrid, we take the output of the **Get-Service** cmdlet and turn it into a .NET array that can then be loaded into the grid. Figure 9-4 shows the resulting form.



| | Status | Name | DisplayName |
|---|---------|----------------------|----------------------|
| ▶ | Stopped | Alerter | Alerter |
| | Stopped | ALG | Application Layer... |
| | Stopped | AppMgmt | Application Mana... |
| | Stopped | aspnet_state | ASP.NET State ... |
| | Running | AudioSrv | Windows Audio |
| | Running | Avg7AIt | AVG7 Alert Mana... |
| | Running | Avg7UpdSvc | AVG7 Update Se... |
| | Running | AVGEMS | AVG E-mail Scan... |
| | Stopped | BAsfipM | Broadcom ASF I... |
| | Running | BITS | Background Intell... |
| | Stopped | Browser | Computer Browser |
| | Stopped | CiSvc | Indexing Service |
| | Stopped | ClipSrv | ClipBook |
| | Stopped | clr_optimization_... | .NET Runtime O... |
| | Running | COMSysApp | COM+ System Ap... |
| | Running | CryptSvc | Cryptographic Se... |
| | Running | CVPND | Cisco Systems, In... |
| | Running | DcomLaunch | DCOM Server Pr... |
| | Running | Dhcp | DHCP Client |
| | Stopped | dmadmin | Logical Disk Man... |

Figure 9-4: PowerShell Grid Form

In this chapter we spent a great deal of time exploring how you can format, group, sort, export, and display PowerShell data. Now you can either run a cmdlet or take a cmdlet or expression, sort and group the data, and save it to an HTML file. PowerShell can also be used to display information in graphical forms using Windows Script Host objects or .NET forms.