Programmer to Programmer™

Professional Windows® PowerShell

Andrew Watt

Wrox

Updates, source code, and Wrox technical support at www.wrox.com

Professional
Windows® PowerShell

Andrew Watt

Wiley Publishing, Inc.

Professional
Windows® PowerShell

Andrew Watt

Wiley Publishing, Inc.

Professional Windows® PowerShell

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published simultaneously in Canada

ISBN: 978-0-471-94693-9

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data

Watt, Andrew, 1953-
Professional Windows PowerShell/Andrew Watt.
p. cm.
ISBN 978-0-471-94693-9 (paper/website)
1. Microsoft Windows (Computer file) 2. Operating systems
(Computers) I. Title.
QA76.76.063W39165 2007
005.4'46--dc22

2007008105

To the memory of my late father George Alec Watt.

To Jonathan, Stephen, Hannah, Jeremy, Peter, and Naomi, each a very special human being to me.

Executive Editor
Chris Webb

Senior Development Editor
Tom Dinse

Technical Editors
Thomas Lee
Joel Stidley

Production Editor
Christine O'Connor

Copy Editor
Foxxe Editorial Services

Editorial Manager
Mary Beth Wakefield

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Joseph B. Wikert

Graphics and Production Specialists
Jennifer Mayberry
Barbara Moore
Alicia B. South
Ronald Terry

Quality Control Technicians
John Greenough
Melanie Hoffman

Project Coordinator
Jennifer Theriot

Proofreading and Indexing
Aptara

Anniversary Logo Design
Richard Pacifico

About the Author

Andrew Watt was on the Windows PowerShell beta program for almost two years before product release. He is a Microsoft Most Valuable Professional, MVP, for SQL Server and is an independent consultant and experienced computer book author. He wrote his first programs in BASIC and 6502 Assembler in 1984 while researching his doctoral thesis.

He is a regular visitor to the Windows PowerShell newsgroup, `microsoft.public.windows.powershell`. He can be contacted by email at SVGDeveloper@aol.com.

Contents

Introduction xv
Acknowledgments xx

Part I: Finding Your Way Around Windows PowerShell 1

Chapter 1: Getting Started with Windows PowerShell 3

Installing Windows PowerShell 3
Installing .NET Framework 2.0 4
Installing Windows PowerShell 7
Starting and Stopping PowerShell 8
Starting PowerShell 8
Exiting PowerShell 10
Startup Options 10
Finding Available Commands 11
Getting Help 14
Basic Housekeeping 17
Case Insensitivity 18
What You Get in PowerShell 19
Interactive Command Shell 19
Cmdlets 20
Scripting Language 21
Summary 23

Chapter 2: The Need for Windows PowerShell 25

Limitations of CMD.exe 27
Batch Files 28
Inconsistency of Implementation 28
Inability to Answer Questions 29
Lack of Integration with GUI Tools 29
The GUI Emphasis in Windows 29
Previous Attempted Solutions 29
Windows Script Host 30
Windows Management Instrumentation 30
Summary 31

Chapter 3: The Windows PowerShell Approach 33

A New Architecture 33
.NET Framework-Based Architecture 34
Object-Based Architecture 35
A New Cross-Tool Approach 38
GUI Shell (MMC Layered over PowerShell) 39
Command Line 39
Command Scripting 41
COM Scripting 43
Namespaces as Drives 45
File System Provider 47
Registry 47
Aliases 48
Variables 49
Active Directory 50
Certificates 51
Extensibility and Backward Compatibility 51
Aliases 51
Use Existing Utilities 53
Use Familiar Commands 55
Long Term Roadmap: Complete Coverage in 3 to 5 Years 55
COM Access 56
WMI Access 56
.NET Class Access 56
Object-Based Approach in PowerShell 56
Object-Based Pipelines 56
A Consistent Verb-Noun Naming Scheme 57
Coping with a Diverse World 58
Upgrade Path to C# 58
Working with Errors 58
Debugging in PowerShell 59
Additional PowerShell Features 59
Extended Wildcards 59
Automatic Variables 60
Summary 62

Chapter 4: Using the Interactive Shell 63

Windows PowerShell's Two Command Line Parsing Approaches 63
Expression Mode Examples 65
Command Mode Examples 66
Mixing Expressions and Commands 69

Exploring a Windows System with Windows PowerShell **69**
Finding Running Processes 69
Filtering Processes Using where-object 71
Filtering Processes Using Wildcards 72
Finding Out about Services 73
Finding Running Services 74
Finding Other Windows PowerShell Commands 75
Using Abbreviated Commands **76**
Command Completion 76
Aliases 77
Working with Object Pipelines **78**
Sequences of Commands 78
Filtering Using where-object 79
Sorting 81
Grouping 83
Pros and Cons of Verbosity **85**
Interactive 85
Stored Commands 87
Summary **87**

Chapter 5: Using Snapins, Startup Files, and Preferences **89**

Startup **89**
Snapins 90
Profiles **97**
Profile.ps1 98
Aliases **100**
The export-alias Cmdlet 105
The get-alias Cmdlet 107
The import-alias Cmdlet 108
The new-alias Cmdlet 108
The set-alias Cmdlet 109
The Help Alias 111
Command Completion 112
Prompts **113**
Preference Variables **115**
Summary **116**

Chapter 6: Parameters **117**

Using Parameters **118**
Finding Parameters for a Cmdlet 121
Named Parameters 124

Wildcards in Parameter Values 125
Positional Parameters 127
Common Parameters 132
Using Variables as Parameters 133
Summary 135

Chapter 7: Filtering and Formatting Output 137

Using the where-object Cmdlet 137
Simple Filtering 138
Using Multiple Tests 140
Using Parameters to where-object 142
The where-object Operators 144
Using the select-object Cmdlet 144
Selecting Properties 145
Expanding Properties 146
Selecting Unique Values 147
First and Last 148
Default Formatting 151
Using the format-table Cmdlet 155
Using the property Parameter 156
Using the autosize Parameter 157
Hiding Table Headers 158
Grouping Output 158
Specifying Labels and Column Widths 159
Using the format-list Cmdlet 161
Using the update-formatdata and update-typedata Cmdlets 162
Summary 163

Chapter 8: Using Trusting Operations 165

Look Before You Leap 166
Using the remove-item Cmdlet 166
Using the whatif Parameter 175
Using the stop-process Cmdlet 175
Using the stop-service Cmdlet 178
Using the confirm Parameter 180
Using the verbose Parameter 181
Summary 182

Chapter 9: Retrieving and Working with Data 183

Windows PowerShell Providers 183
Using the get-psdrive Cmdlet 184
Using the set-location Cmdlet 188
Using the passthru Parameter 190
Using the get-childitem Cmdlet 191
Using the get-location Cmdlet 194
Using the get-content Cmdlet 196
Using the measure-object Cmdlet 201
The new-item Cmdlet 203
The new-psdrive Cmdlet 204
Summary 205

Chapter 10: Scripting with Windows PowerShell 207

Enabling Scripts on Your Machine 207
Using the read-host Cmdlet 212
Using the write-host Cmdlet 214
The Arithmetic Operators 218
Operator Precedence 219
The Assignment Operators 220
The Comparison Operators 222
The Logical Operators 225
The Unary Operators 226
Using the set-variable and Related Cmdlets 227
The set-variable Cmdlet 228
The new-variable Cmdlet 229
The get-variable Cmdlet 230
The clear-variable Cmdlet 231
The remove-variable Cmdlet 232
Summary 234

Chapter 11: Additional Windows PowerShell Language Constructs 235

Arrays 235
Creating Typed Arrays 239
Modifying the Structure of Arrays 241
Working from the End of Arrays 245
Concatenating Arrays 248
Associative Arrays 249

Conditional Expressions **250**
The if Statement 251
The switch Statement 254
Looping Constructs **256**
The for Loop 256
The while Loop 258
The do/while Loop 259
The foreach Statement 260
Summary **262**

Chapter 12: Processing Text **263**

The .NET String Class **263**
Working with String Methods 267
Casting Strings to Other Classes **287**
URI 287
datetime 288
XML 289
Regex 289
Summary 291

Chapter 13: COM Automation **293**

Using the new-object Cmdlet **293**
Working with Specific Applications **294**
Working with Internet Explorer 294
Working with Windows Script Host 299
Working with Word 301
Working with Excel 302
Accessing Data in an Access Database 303
Working with a Network Share 305
Using Synthetic Types **306**
Summary **308**

Chapter 14: Working with .NET **309**

Windows PowerShell and the .NET Framework **309**
Creating .NET Objects **311**
The new-object Cmdlet 311
Other Techniques to Create New Objects 317
Inspecting Properties and Methods **320**
Using the get-member Cmdlet 320

Using .NET Reflection **324**
Using the GetMembers() Method 324
Using the GetMember() Method 326
Using the GetMethods() Method 328
Using the GetMethod() Method 329
Using the GetProperties() Method 330
Using the GetProperty() Method 331
Using System.Type Members 333
Summary **334**

Part II: Putting Windows PowerShell to Work **335**

Chapter 15: Using Windows PowerShell Tools for Discovery **337**

Exploring System State **338**
Using the get-location Cmdlet 338
Handling Errors 345
Namespaces 346
PowerShell Aliases 346
PowerShell Functions and Filters 349
PowerShell Variables 350
Exploring the Environment Variables **351**
Exploring the Current Application Domain **353**
Exploring Services **357**
Using the get-service Cmdlet **358**
Using the new-service Cmdlet 360
Using the restart-service Cmdlet 361
Using the set-service Cmdlet 362
Using the start-service Cmdlet 362
Using the stop-service Cmdlet 363
Using the suspend-service Cmdlet 364
Summary **365**

Chapter 16: Security **367**

Minimizing the Default Risk **368**
The Certificate Namespace **374**
Signed Scripts **376**
Creating a Certificate 376
The set-authenticodesignature Cmdlet 377
The get-authenticodesignature Cmdlet 379
Summary **379**

Chapter 17: Working with Errors and Exceptions 381

Errors in PowerShell 381
$Error 383
Using Error-Related variables 388
Using the $ErrorView variable 389
Using the $ErrorActionPreference variable 390
Trap Statement 392
Using Common Parameters 397
Using the ErrorAction Parameter 397
Using the ErrorVariable Parameter 399
The write-error Cmdlet 400
Summary 401

Chapter 18: Debugging 403

Handling Syntax Errors 403
The set-PSDebug Cmdlet 408
The write-debug Cmdlet 413
Tracing 418
The trace-command Cmdlet 419
The set-tracesource Cmdlet 422
The get-tracesource Cmdlet 422
Summary 423

Chapter 19: Working with the File System 425

Path Names in Windows PowerShell 426
Fully Qualified Path Names 427
Relative Path Names 430
Path Names and Running Commands 431
Simple Tasks with Folders and Files 434
Finding the drives on a system 434
Finding Folders and Files 434
Finding File Characteristics 436
Exploring Files Using the select-object Cmdlet 439
Finding Hidden Files 442
Tab Completion 443
Redirection 445
Creating Custom Drives 447
Cmdlets for File Actions 449
Using the out-file Cmdlet 449

Using Cmdlets to Work with Paths **450**
Summary **453**

Chapter 20: Working with the Registry **455**

Introduction to the Registry **455**
Exploring the Registry Using Windows PowerShell **458**
Selecting a Hive 458
Navigating to a Desired Key 459
Changing the Registry **461**
Summary **464**

Chapter 21: Working with Environment Variables **465**

Environment Variables Overview **465**
The Environment Command Shell Provider **468**
Exploring Environment Variables **470**
Modifying Environment Variables **471**
Summary **473**

Part III: Language Reference **475**

Chapter 22: Working with Logs **477**

Event Log Basics **477**
The get-eventlog Cmdlet **480**
Summary **494**

Chapter 23: Working with WMI **495**

Introducing Windows Management Instrumentation **496**
Managed Resources 496
WMI Infrastructure 497
CIM Object Manager 498
The CIM Repository 499
WMI Consumers 499
WMI Tools 499
Using the get-wmiobject Cmdlet **502**
Finding WMI Classes and Members 506
Exploring a Windows System **509**
Characterizing the CPU 509
Finding Memory 510

Exploring Services 512
Exploring Remote Machines 513
Summary **514**

Index **515**

Introduction

Windows PowerShell version 1.0 is Microsoft's first step towards a radically new, exciting, powerful, and comprehensive command line administration tool for Microsoft Windows. For years Windows users have had to use a very limited command line shell, `CMD.exe`. But no more! Windows PowerShell introduces a new, more powerful, more flexible, more consistent object-based command line tool and scripting language (with a highly consistent syntax). PowerShell is specifically designed to make it possible for you to do administrative tasks that previously you couldn't do at all from the command line and to make many familiar administrative tasks easier. Windows PowerShell can be installed on any machine that is running Windows Server 2003 (Service Pack 1 or later), Windows XP (Service Pack 2 or later) or Windows Vista.

Windows PowerShell is based on cmdlets (pronounced commandlets), which are small, modular commands consistently based on a *verb-noun* naming system. For example the `get-process` cmdlet retrieves information about running processes on a machine. You can flexibly combine cmdlets in *pipelines* to create custom functionality. Pipelines can be simple or of arbitrary complexity. You choose how to combine cmdlets to do things that are useful for you. The sky's the limit, pretty much.

It's great to be able to run pipelines from the command, line but once you have worked on a complex pipeline so that it does exactly what you want, you won't want to discard it. In Windows PowerShell, you can save your code as *scripts* (using the same syntax you used on the command line) and run those scripts from the command line when needed.

Who This Book Is For

This book is intended to help you get up to speed with Windows PowerShell whether you administer one Windows machine or many thousands. Although the book is in the Wrox Professional series I don't assume that you have any previous experience using Windows PowerShell since, for most readers, your previous experience of PowerShell 1.0 is likely to be zero or minimal. On the other hand, I assume you are familiar with many basics of how Windows works and generally don't spend much time telling you about basic Windows functionality outside PowerShell.

I show you how to use many of the cmdlets available in Windows PowerShell 1.0 and show you how you can combine cmdlets to create pipelines. I show you how to store your code as scripts and how to run them.

What This Book Covers

First I spend a little time introducing you to why Windows PowerShell has been created. I look briefly at how previous Microsoft technologies attempted to help you administer Windows computers, then look at how Windows PowerShell brings its new and more powerful solutions to existing challenges.

I show you how to use PowerShell from the command line, initially using individual cmdlets to carry out fairly simple tasks. Then I show you how to construct pipelines to carry out more complex tasks. I then show you how to use *parameters* to modify the behavior of individual cmdlets. And, of course, how you can combine cmdlets and their parameters in useful pipelines.

Windows PowerShell can, at times, produce almost unmanageable amounts of information. I show you techniques that help you to filter output from commands and how to present the data of interest onscreen.

Once you have mastered the basics of PowerShell, you will want to store your code in script files. I show you how to store and run scripts and describe and demonstrate many features of the PowerShell language.

In the latter part of the book I show you how to use PowerShell to carry out various tasks. I show you how to use PowerShell to work with text, to automate COM objects and to script .NET, I show you how to set security for Windows PowerShell and how to make use of PowerShell tools to help you debug your code.

In the final chapters I show you how you can use PowerShell to work with files, the registry, environment variables, and logs.

Throughout the book I describe the functionality of many individual cmdlets and show you how you can use many combinations of cmdlets and parameters.

This book doesn't attempt to provide comprehensive coverage of what Windows PowerShell can do. In fact, no book can do that since there is essentially an infinite number of ways to combine PowerShell cmdlets. The important thing that I have tried to achieve is to show you how to combine the parts available to you in PowerShell so that you can go on to combine them in the ways that makes most sense for your needs. However, I intend to cover topics that I couldn't include in this book in a blog at `www.propowershell.com`. I hope to have the site up and running by the time this book is in print. If you want particular topics to be discussed or demonstrated on the blog contact me through that site and I will, time permitting, cover the additional topics most frequently requested.

How This Book Is Structured

I have summarized the content of this book in the preceding section. In this section, I briefly suggest how you might want to use this book depending on your level of experience with PowerShell.

Most readers will come to this book with minimal experience with PowerShell. Therefore, I have written it so that you can read it from beginning to end, as an extended tutorial if you wish. If you're completely new to PowerShell that is probably the best way to use the book.

On the other hand, if you already have some experience with PowerShell the Contents and Index allow you to dip into chapters or sections that are particularly suitable to your needs, as summarized in the preceding section of this Introduction.

What You Need to Use This Book

To run Windows PowerShell, you need to have a compatible version of Microsoft Windows installed. Specifically, you need Windows Server 2003 (Service Pack 1 or later), Windows XP (Service Pack 2 or later) or Windows Vista.

In addition, before you install and run Windows PowerShell you need to install the .NET Framework version 2.0. Initial experience with version 3.0 of the .NET Framework suggests that Windows PowerShell 1.0 also works well with it.

I anticipate that Windows PowerShell will also run on other future versions of Windows, including the server operating system that is currently codenamed Longhorn Server. However, at the time of writing, I have not had the opportunity to text PowerShell 1.0 on Longhorn Server.

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

> **Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.**

Tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show filenames, URLs, and code within the text like this: `persistence.properties`.
- I show you code to type at the command line like this:

```
get-process
```

or, where code is a pipeline which extends over two or more lines, like this:

```
get-process |
format-table
```

- We present code in two different ways:

```
In code examples we highlight new and important code with a gray background.
```

```
The gray highlighting is not used for code that's less important in the present
context, or has been shown before.
```

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at www.wrox.com. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists), and click the Download Code link on the book's detail page to obtain all the source code for the book.

Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-0-471-94693-9.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to www.wrox.com and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page, you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at www.wrox.com/misc-pages/booklist.shtml.

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At http://p2p.wrox.com, you will find a number of different forums that will help you not only as you read this book but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to `p2p.wrox.com` and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

In addition to the facility at `p2p.wrox.com` I hope to provide content to complement this book in a blog at `www.propowershell.com`. I hope to have the site up and running by the time this book is in print.

Acknowledgments

Any complex task, including the creation and production of a computer book, is a team effort and I would like to thank several individuals who have helped me produce this book. The book has been a long time in gestation in part due to the timescale of the development of Windows PowerShell 1.0 but in part due to pressures on my time around important milestones. I would like to thank everyone for their patience through a long process.

First, I would like to thank Chris Webb, Executive Editor at Wrox who invited me to write the book.

I would also like to thank Tom Dinse, senior development editor at Wrox, who helped mold my draft text in useful ways, made many helpful suggestions along the way and kept me right about series guidelines.

The technical editors had a hard time, since I wrote several different drafts against intermediate development builds of PowerShell. Over the months, the development team at Microsoft made many changes to help improve the product, but each time they changed cmdlet and parameter names there was a whole series of changes across multiple chapters to be identified by the author and technical editors. I would particularly like to thank Joel Stidley whose comments on later drafts were invariably pertinent and who picked up a good number of changes that I had missed as I went through chapters to reflect changes in the final release of Windows PowerShell 1.0. Thomas Lee did a manful job of working with earlier drafts. Any remaining errors or omissions are my own responsibility.

Part I

Finding Your Way Around Windows PowerShell

Chapter 1: Getting Started with Windows PowerShell

Chapter 2: The Need for Windows PowerShell

Chapter 3: The Windows PowerShell Approach

Chapter 4: Using the Interactive Shell

Chapter 5: Using Snapins, Startup Files, and Preferences

Chapter 6: Parameters

Chapter 7: Filtering and Formatting Output

Chapter 8: Using Trusting Operations

Chapter 9: Retrieving and Working with Data

Chapter 10: Scripting with Windows PowerShell

Chapter 11: Additional Windows PowerShell Language Constructs

Chapter 12: Processing Text

Chapter 13: COM Automation

Chapter 14: Working with .NET

1

Getting Started with Windows PowerShell

If you are like me, then when you begin to look seriously at an interesting piece of software, you like to get your hands dirty and play with it from the beginning. In this chapter, I show you how to get started using Windows PowerShell, and I'll show you enough of the PowerShell commands to let you begin to find your way around effectively. In the rest of the book, I help you build on that initial knowledge so that you can use PowerShell for a wide range of useful tasks, depending on your requirements.

Windows PowerShell, as you probably already know, is Microsoft's new command shell and scripting language. It provides a command line environment for interactive exploration and administration of computers, and by storing and running Windows PowerShell commands in a script file, you can run scripts to carry out administrative tasks multiple times. Windows PowerShell differs in detail from existing command line environments on the Windows and Unix platforms, although it has similarities to past environments. In Chapter 3, in particular, I explain more about the PowerShell approach, although differences from existing command shells and scripting languages will emerge in every chapter.

Once you have had a brief taste of PowerShell, you will need to understand a little of the assumptions and approach that lie behind the design decisions that have made PowerShell the useful tool that it is. In Chapter 2, I step back from using the PowerShell command line and look at the strengths and deficiencies of some existing Microsoft approaches to system management and then, in Chapter 3, take a look at the philosophy and practical thought that lies behind the approach taken in Windows PowerShell.

Installing Windows PowerShell

Windows PowerShell depends on the presence of the .NET Framework 2.0. Before you install PowerShell, you need to be sure that you have the .NET Framework 2.0 installed.

Installing .NET Framework 2.0

At the time of writing, the 32-bit version of the .NET Framework 2.0 runtime is available for downloading from `www.microsoft.com/downloads/details.aspx?FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en`.

If you are using 64-bit Itanium processors, download the .NET Framework 2.0 runtime from `www.microsoft.com/downloads/details.aspx?familyid=53C2548B-BEC7-4AB4-8CBE-33E07CFC83A7&displaylang=en`. Windows PowerShell is only available on Windows Server 2003 for Itanium processors.

If you are using AMD 64-bit processors, download the runtime from `www.microsoft.com/downloads/info.aspx?na=47&p=3&SrcDisplayLang=en&SrcCategoryId=&SrcFamilyId=F4DD601B-1B88-47A3-BDC1-79AFA79F6FB0&u=details.aspx%3ffamilyid%3dB44A0000-ACF8-4FA1-AFFB-40E78D0788B00%26displaylang%3den`.

If you are unsure whether or not you have .NET Framework 2.0 installed, navigate to `C:\Windows\Microsoft.NET\Framework` *(if necessary substitute another drive letter if your system drive is not drive C:). In that folder you will find folders that contain the versions of the .NET Framework that are installed on your machine. If you see a folder named v2.0.50727, then you have the .NET Framework 2.0 installed. The .NET Framework 2.0 SDK, which you can download separately, is useful as a source of information on .NET 2.0 classes that you can use with PowerShell.*

If you want to install the 32 bit .NET Framework 2.0 Software Development Kit (SDK), download it from `www.microsoft.com/downloads/details.aspx?FamilyID=fe6f2099-b7b4-4f47-a244-c96d69c35dec&displaylang=en`**. To install the .NET Framework 2.0 SDK, you must first install the 32-bit runtime.**

There are also 64-bit versions of the .NET Framework 2.0 SDK available for downloading. The version of the runtime for Itanium is located at `www.microsoft.com/downloads/details.aspx?familyid=F4DD601B-1B88-47A3-BDC1-79AFA79F6FB0&displaylang=en`**. The 64-bit version for AMD processors is located at** `www.microsoft.com/downloads/details.aspx?familyid=1AEF6FCE-6E06-4B66-AFE4-9AAD3C835D3D&displaylang=en`**.**

Figure 1-1 shows what you would expect to see in the Framework folder on a clean install of Windows 2003 Service Pack 1 which does not have the .NET Framework 2.0 runtime installed.

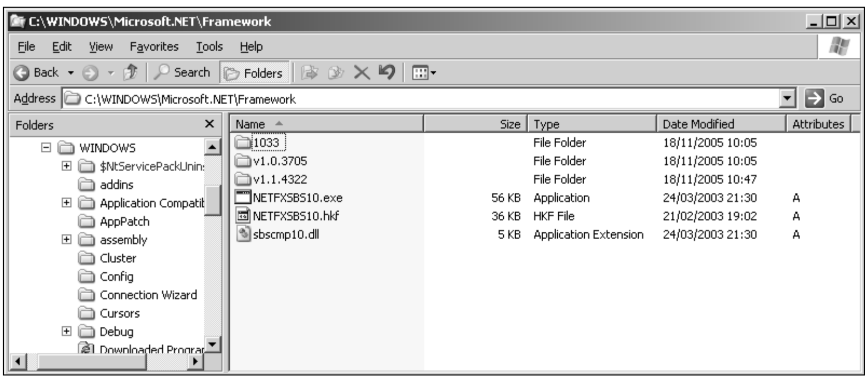


Figure 1-1

Figure 1-2 shows the appearance of the Framework folder on a clean install of Windows 2003 Service Pack 1 after the .NET Framework 2.0 runtime has been installed.

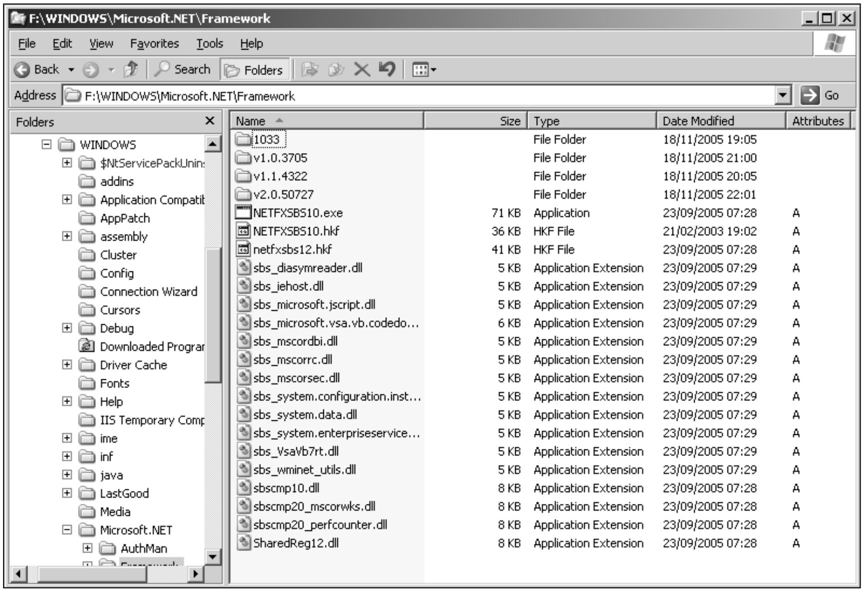


Figure 1-2

You don't need to delete the v1.0.3705 or v1.1.4322 folders. In fact, you are likely to cause problems for applications that need earlier versions of the .NET Framework if you delete those folders. The .NET Framework 2.0 is designed to run side by side with .NET Framework 1.0 and 1.1.

To install the .NET Framework 2.0, follow these steps.

1. Navigate in Windows Explorer to the folder where you downloaded the installer, `dotnetfx.exe`.
2. Double-click the installer. On the splash screen that opens, click Next.
3. On the EULA screen, accept the license agreement and click Install.
4. The installer then proceeds to install the .NET Framework 2.0, as shown in Figure 1-3.
5. When the installation has completed successfully, you should see a screen similar to Figure 1-4.
6. If you have Internet connectivity, click the Product Support Center link shown in Figure 1-4 to check for any updates.

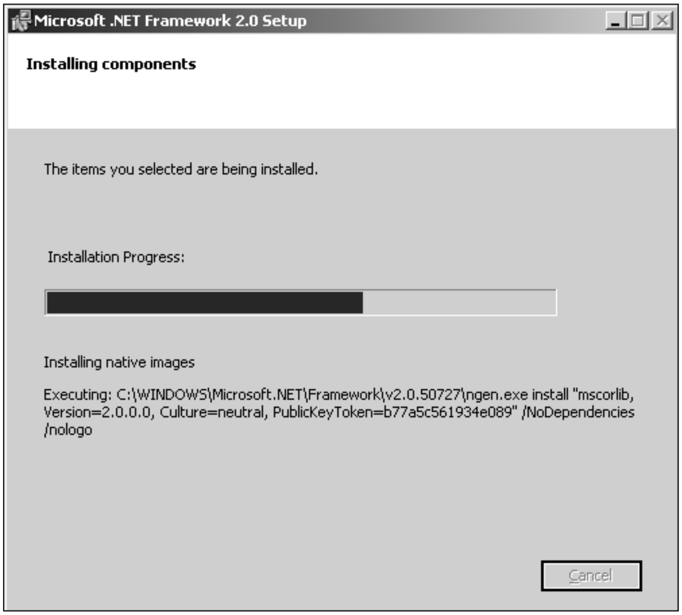


Figure 1-3

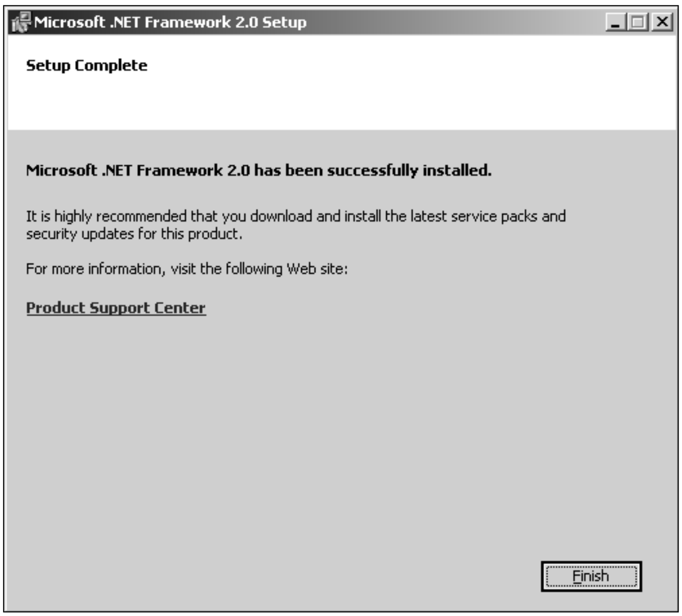


Figure 1-4

Once you have installed the .NET Framework 2.0, you can then install Windows PowerShell.

Installing Windows PowerShell

To install Windows PowerShell on a 32-bit system, follow these steps. If you are installing it on a 64-bit system, the installer filename will differ.

1. Double-click the `.exe` installer file appropriate for the version of Windows PowerShell you want to install. The initial screen of the installation wizard, similar to the one shown in Figure 1-5, is displayed.

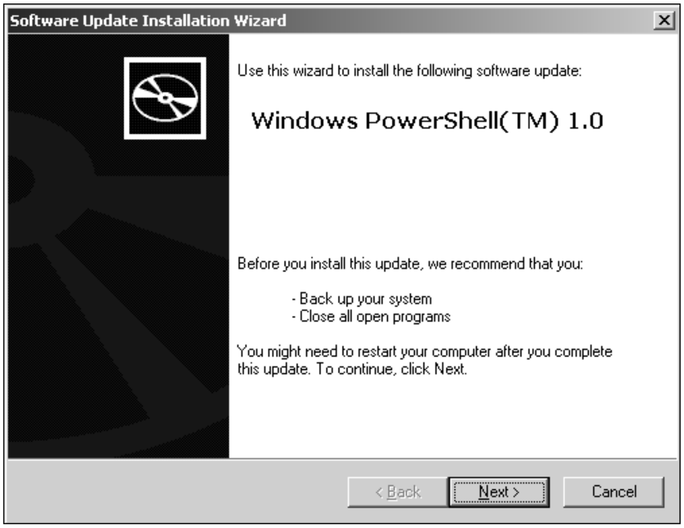


Figure 1-5

2. Click Next.
3. Accept the license agreement and click Next.
4. If you are installing on drive C: on a 32-bit system, the default install location is `C:\Windows\System32\windowspowershell\v1.0`.
5. When the installation has completed successfully you will see a screen similar to Figure 1-6.
6. Click Finish.

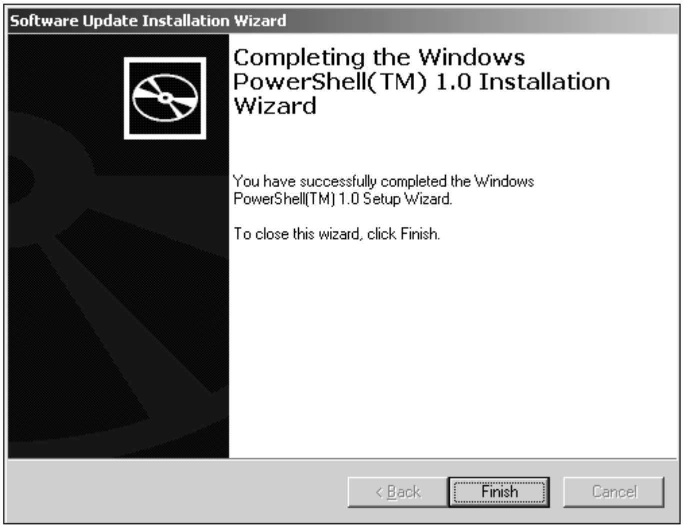


Figure 1-6

Starting and Stopping PowerShell

Once you have installed Windows PowerShell, you have several options for starting it.

Starting PowerShell

To start PowerShell without using any profile file to customize its behavior, open a command window (On Windows 2003, select Start ➪ All Programs ➪ Accessories ➪ Command Prompt), then type:

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\powershell.exe -NoProfile
```

After a short pause, the Windows PowerShell prompt should appear (see Figure 1-7).

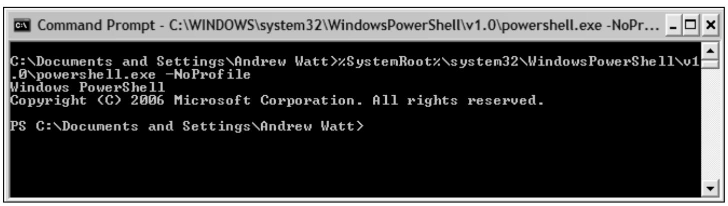


Figure 1-7

If you are still using a Release Candidate and attempt to start PowerShell by simply typing `PowerShell.exe` at the command shell prompt, you may see the error message shown in Figure 1-8. To fix that, update to the final release version.

Figure 1-8

Alternatively, you can start PowerShell by selecting Start ➪ All Programs ➪ Windows PowerShell 1.0 ➪ Windows PowerShell (see Figure 1-9).

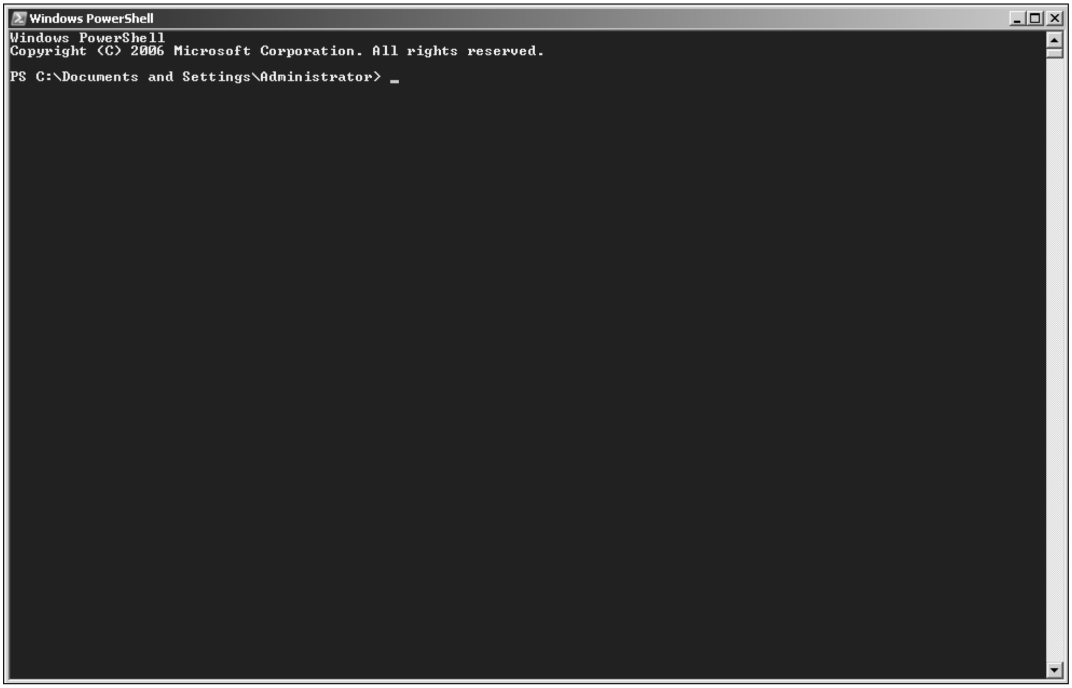


Figure 1-9

Because of security concerns about previous Microsoft scripting technologies, the default setting of Windows PowerShell is that scripting is locked down. Specifically, when Windows PowerShell starts, it does not attempt to run profile files (which are PowerShell scripts) that contain various settings controlling how PowerShell should run. Whichever way you start PowerShell initially, you will probably later want to enable scripts. To do that, you use the `set-executionpolicy` cmdlet. Type:

```
set-executionpolicy -ExecutionPolicy "RemoteSigned"
```

and you will be able to run locally created scripts without signing them. I cover execution policy in more detail in Chapter 10.

There are several additional options for starting PowerShell, and I will briefly describe all of those — after I show you how to stop PowerShell.

Exiting PowerShell

To stop PowerShell, simply type the following at the PowerShell command line:

```
Exit
```

and you are returned to the `CMD.exe` command prompt (assuming that you started PowerShell from the `CMD.exe` prompt). If you started PowerShell using Start ➪ All Programs ➪ Windows PowerShell 1.0 ➪ Windows PowerShell, the PowerShell window closes.

You can't use "`quit`" to exit PowerShell. It just causes an error message to be displayed.

Startup Options

You have several options for how you start PowerShell. These are listed and explained in the following table. On the command line, each parameter name is preceded by a minus sign.

Parameter	Explanation
`Command`	The value of the `Command` parameter is to be executed as if it were typed at a PowerShell command prompt.
`Help`	Displays information about the startup options for PowerShell summarized in this table.
`InputFormat`	Specifies the format of any input data. The options are "Text" and "XML."
`NoExit`	Specifies that PowerShell doesn't exit after the command you enter has been executed. Specify the `NoExit` parameter before the `Command` parameter.
`NoLogo`	The copyright message usually displayed when PowerShell starts is omitted. Specifying this parameter causes the copyright message not to be displayed.
`NonInteractive`	Use this parameter when no user input is needed nor any output to the console.
`NoProfile`	The user initialization scripts are not run.
`OutputFormat`	Specifies the format for outputting data. The options are "Text" and "XML."
`PSConsoleFile`	Specifies a Windows PowerShell console file to run at startup.

To view information about all help options, type:

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\powershell.exe -Help
```

or:

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\powershell.exe -?
```

or:

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\powershell.exe /?
```

at the command line. Each of those commands will cause the information, part of which is shown in Figure 1-10, to be displayed.

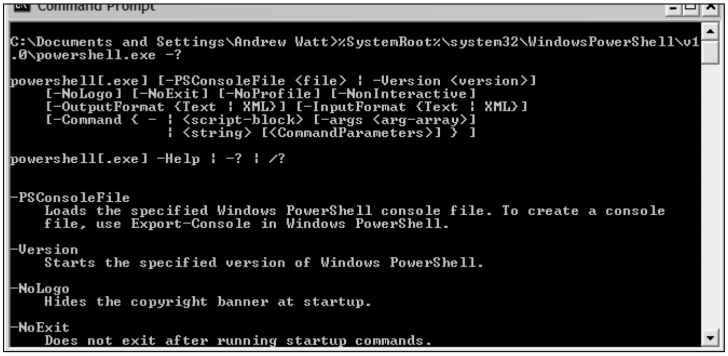


Figure 1-10

Notice that there are sets of parameters that you can use with `PowerShell.exe`. You can combine parameters only in the ways shown in Figure 1-10.

The preceding commands give you help on how to start Windows PowerShell. Once you start PowerShell, you also need to know where to find help on individual PowerShell commands. As a first step, you need to be able to find out what commands are available to you.

Individual PowerShell commands are small and granular. As a result, they are called cmdlets (pronounced "commandlets").

Finding Available Commands

In this section, I will show you a few commonly used commands and show you how to explore the PowerShell cmdlets to see what PowerShell commands are available on your system. The `get-command` cmdlet allows you to explore the commands available to you in Windows PowerShell.

The simplest, but not the most useful, way to use the `get-command` cmdlet is simply to type:

```
get-command
```

at the PowerShell command line. Several screens of command names scroll past when you do this — there are a lot of cmdlets in PowerShell. It's more useful to view the information one screen at a time. You achieve that by typing:

```
get-command | More
```

at the PowerShell command line. The result is similar to that shown in Figure 1-11. If you run that command and carefully read the available commands, you will get some idea of the scope of functionality that PowerShell allows you to control and manage.

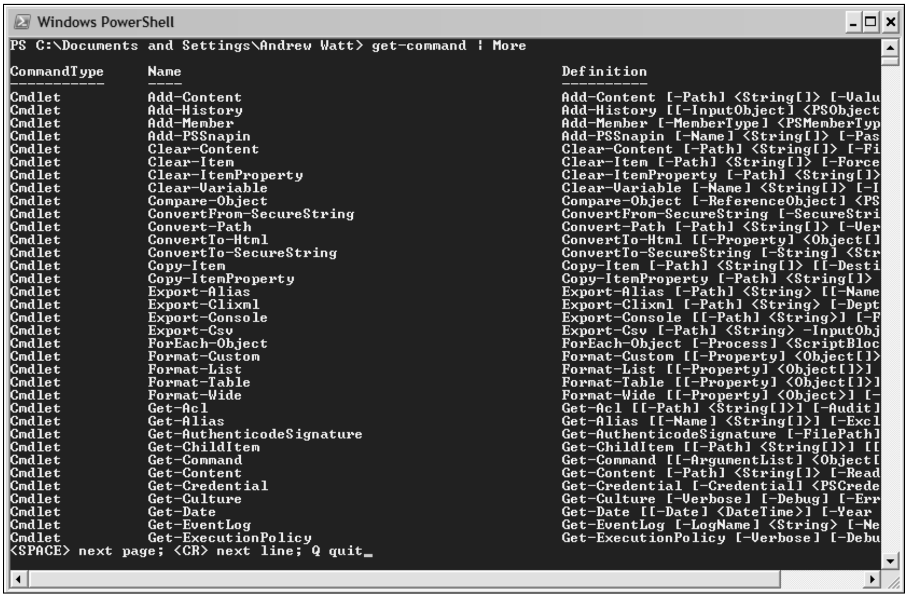


Figure 1-11

To view another screen of commands, press the spacebar once. Repeat this to view each additional screen of commands.

> *PowerShell commands are formed of a verb, followed by a hyphen (or minus sign), followed by a noun. The* `get-command` *cmdlet illustrates the structure. The verb "*`get`*" is followed by a hyphen, which is followed by a noun "*`command`*." PowerShell uses the singular form of the noun, even when, as is often the case, you want to find multiple items that satisfy your requirements. Thus, you might use* `get-process` *to get all the processes running on a system, as opposed to* `get-processes`*.*

You can use wildcards to focus your search for the relevant command. For example, to find all commands that use the `get` verb, use the following command:

```
get-command get-*
```

or, the slightly tidier:

```
get-command get-* | More
```

The argument to the `get-command` cmdlet uses the `*` wildcard. The argument `get-*` finds any command whose name begins with get, a hyphen, and zero or more other characters. As you can see in Figure 1-12, there are many cmdlets that use the `get` verb.

Other verbs worth looking for include `add`, `format`, `new`, `set`, and `write`. To see a complete list of available verbs, type the following command:

```
get-command | group-object verb
```

Figure 1-13 shows the results. The preceding command uses a pipeline that consists of two steps. The first uses the `get-command` cmdlet to create objects representing all available commands. The second step uses the `group-object` cmdlet to group the results by the verb.

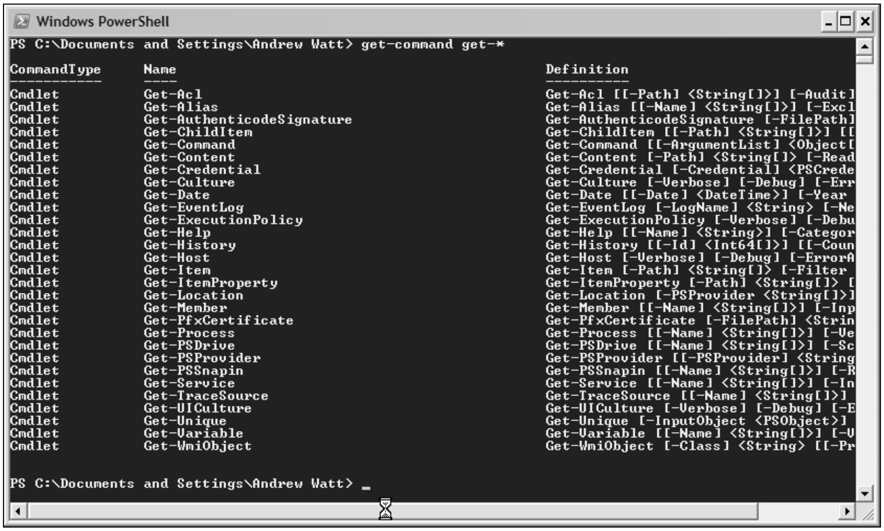


Figure 1-12

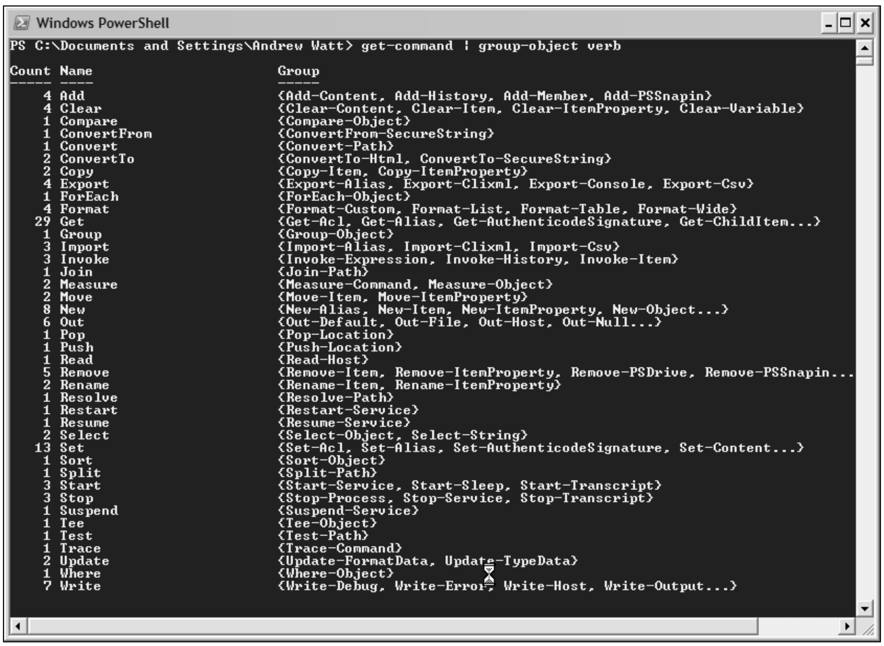


Figure 1-13

To find the nouns available in your installation of PowerShell, use the following command:

```
get-command | group-object noun
```

If you want to sort the nouns alphabetically use the following command:

```
get-command | group-object noun | sort-object name
```

You can also use the `get-command` cmdlet to explore in other ways. For example, suppose that you want to find all cmdlets that you can use to work with processes. The preceding command shows you that `process` is one of the nouns used in cmdlets. One way to display information about all cmdlets that operate on processes is to use the following command:

```
get-command *-process
```

Figure 1-14 shows you that there are only two cmdlets that you can use to work specifically with processes. As you construct pipelines with multiple steps, you have many other cmdlets available for use with process-related information.

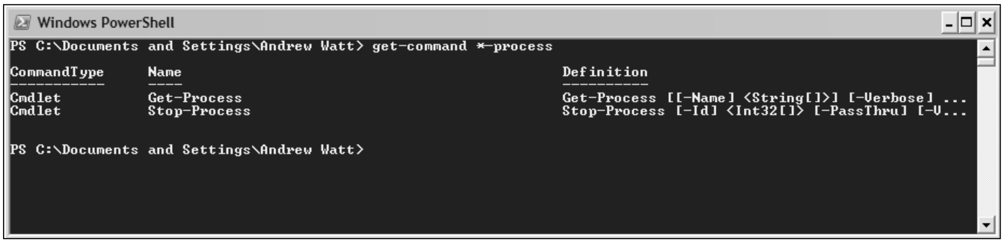


Figure 1-14

You can adapt the preceding command to find cmdlets relevant to other nouns. For example, the command:

```
get-command *-service
```

will find all cmdlets that relate to services.

Getting Help

When you're using PowerShell, you need to be able to find out how to use commands that you are already aware of or that you find by using the techniques described in the previous section.

You use the `get-help` cmdlet to get help information about individual cmdlets. You can use the `get-help` cmdlet with or without parameters. Using the `get-help` cmdlet with no parameters displays abbreviated help information.

For example, to get help on the `get-process` cmdlet type either:

```
get-help get-process
```

or:

```
get-process -?
```

at the PowerShell command line.

The default behavior of the `get-help` cmdlet when providing help information about a specific command is to dump all the help text to the screen at once, causing anything that won't fit on one screen to scroll off the screen and out of sight. You may find it more useful to display the help information one screen at a time by using `More`:

```
get-help get-process | More
```

or:

```
get-process -? | More
```

You are likely to have the `help` function available to you. It behaves similarly to the `get-help` cmdlet, except that the `help` function displays the help information one screen at a time. To display the help information for the `get-process` cmdlet one screen at a time, you can type:

```
help get-process
```

Since that is a little shorter to type than the `get-help` syntax, you may find that it's more convenient.

PowerShell displays help information in a way similar to `man` in Unix. The help for each command or other piece of syntax is structured in the following sections:

- **Name** – The name of the cmdlet
- **Synopsis** – A brief text description of the cmdlet
- **Syntax** – Demonstrates how the cmdlet can be used
- **Detailed Description** – A longer text description of the cmdlet
- **Parameters** – Provides detailed information about how to use each parameter
- **Input Type** – Specifies the type of the input object(s)
- **Return Type** – Specifies the type of the returned object
- **Examples** – Examples of how to use the cmdlet
- **Related Links** – Names of other cmdlets with related functionality
- **Remarks** – Information about using parameters with the cmdlet

For some commands, some sections may contain no help information.

When you use no parameter with the `get-help` cmdlet, you see the following sections of information:

- Name
- Synopsis
- Syntax
- Detailed Description
- Related Links
- Remarks

If you use the `-detailed` parameter, for example:

```
get-help get-process -detailed
```

you see the following sections of help information:

- Name
- Synopsis
- Syntax
- Detailed Description
- Parameters
- Examples
- Remarks

If you use the `-full` parameter, for example:

```
get-help get-process -full
```

you see the following sections of help information:

- Name
- Synopsis
- Syntax
- Detailed Description
- Parameters
- Input Type
- Return Type
- Notes
- Examples
- Related Links

In addition to the built-in help about cmdlets, you can also access help about aspects of the PowerShell scripting language using the `get-help` cmdlet. If you don't know what help files on the language are available, use the command:

```
get-help about_* | more
```

to display them. Figure 1-15 shows one screen of results. This works, since each of these help files begins with `about_`.

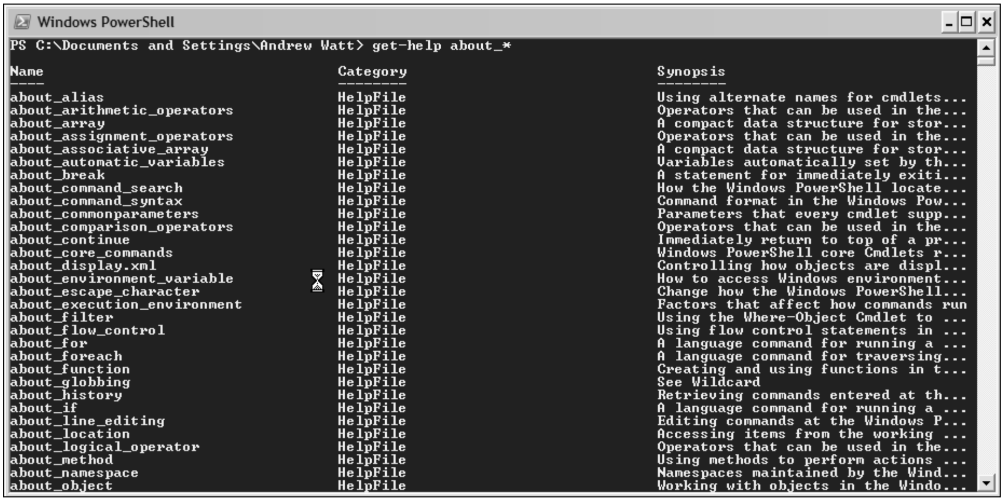


Figure 1-15

An alternative way to explore the available help files for an install on 32-bit hardware is to open Windows Explorer, navigate to the folder `C:\Windows\System32\WindowsPowerShell\v1.0`, and look for text files whose name begins with `about`. If your system drive is not drive C: modify the path accordingly.

Basic Housekeeping

On the surface, a lot of PowerShell works in the same way as `CMD.exe`. In this section, I describe a couple of basic commands that you will likely use frequently.

To clear the screen, you can type:

```
clear-host
```

or:

```
clear
```

or:

```
cls
```

at the PowerShell command line.

To repeat the last-used PowerShell command, press the F3 key once.

To cycle through recently used PowerShell commands, press the up arrow as necessary to move back to the command that you want to reuse or to adapt. You can also use the `get-history` cmdlet to see the command history. By default, you will be able to see the last 64 commands, but if you or an administrator has modified the value of the `$MaximumHistoryCount` variable, the number of commands available in the history may differ.

At the risk of stating the obvious, PowerShell offers you a number of ways to review information that has scrolled out of sight by using the scroll bars in the PowerShell command window. Click in the scroll bar area or drag the slider in the scroll bar area to move up and down through the information in the PowerShell console window.

Case Insensitivity

In PowerShell, cmdlet names are case-insensitive. In general, cmdlet parameter information is generally also case-insensitive, although there are cases where this is not the case.

All PowerShell cmdlet names, in the *verb-noun* form are case-insensitive. Similarly, all named parameters have parameter names that are case-insensitive. For example, to retrieve information about available commands you can use:

```
get-command
```

or:

```
Get-Command
```

or any other variant of the name using mixed case.

The Windows operating system does not consider case significant in filenames. So, any time that you use a filename as an argument to a PowerShell command, case is not significant by default. For example, to redirect the current date and time to a file named `Text.txt` on drive `C:`, use the following command, which includes redirection:

```
get-date > C:\Test.txt
```

The > character is the redirection operator, which redirects output from the screen (the default) to some specified target — in this case, a file on drive `C:`.

An exception to the general rule of no case-sensitivity is when you use class names from the .NET Framework. PowerShell allows you work directly with classes from the .NET Framework. I discuss this in more detail in Chapter 13.

What You Get in PowerShell

On the surface, PowerShell simply appears to be a new command shell, but you get a highly flexible scripting language with it, too. The following sections describe aspects of the PowerShell package and provide some simple examples of how you can use it.

Interactive Command Shell

As I showed you earlier in the chapter, PowerShell comes complete with a range of commands, called cmdlets, that you can use interactively. By combining these commands in pipelines, you can filter, sort, and group objects. Pipelines are a way of combining commands. They have the general form:

```
command1 | command2
```

where each step of the pipeline may contain a PowerShell cmdlet, often using multiple parameters. The | character is used to separate the steps of a pipeline. A pipeline can be of arbitrary length.

In the rest of this book, I demonstrate some of the neat tricks you can use to take advantage of pipelines to manage your system.

The following command is a three-step pipeline that retrieves information about running processes, sorts it by process name, and displays selected parts of the results in a table.

```
get-process svchost |
sort-object ProcessName |
format-table ProcessName, Handlecount
```

As you can see in Figure 1-16, you can type the pipeline on a single line. In this book, I will generally present multistep pipelines on multiple lines, since that makes it easier for you to see what each step of the pipeline is doing.

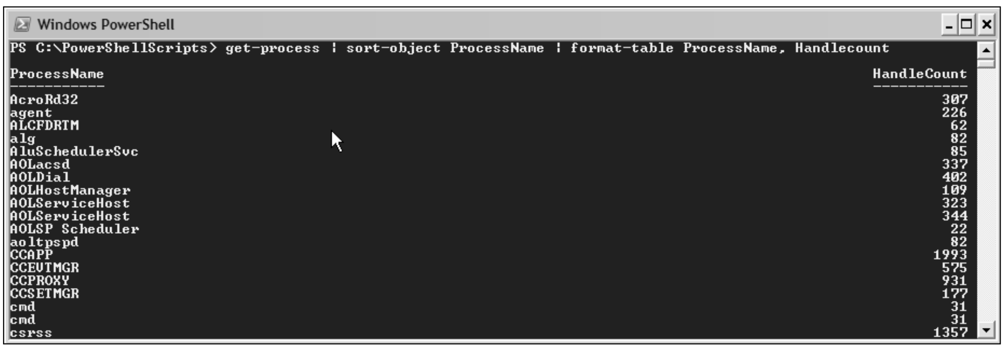


Figure 1-16

If you prefer, you can type each step of multistep pipelines on separate lines on the command line, which I show you in Figure 1-17. Notice that each step of the pipeline except the last ends in the pipe character (|) and that the command prompt changes to >>. After you type the last step of the pipeline, press Return twice and the command will be executed as if you had typed it all on a single line.

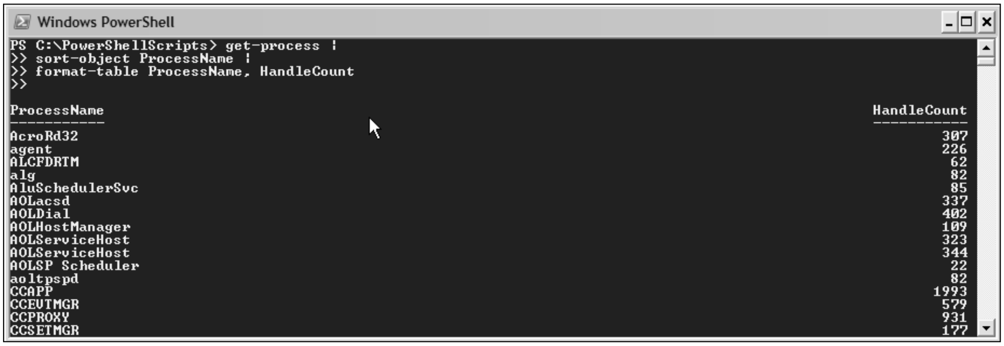


Figure 1-17

Often, you will use the final step of a pipeline to choose how to display information. However, that's not essential, since there is default formatting of output. However, you can use formatting commands as the final step in a pipeline to customize the display and produce a desired output format.

Cmdlets

In a default install of PowerShell version 1, you get more than 100 cmdlets. I look at these individually in more detail in Chapter 4 and later chapters.

If you want to count the number of cmdlets in your version of PowerShell, you can type the following at the command line:

```
$a = get-command;$a.count
```

or:

```
$a = get-command
$a.count
```

or:

```
$(get-command).count
```

The semicolon is the separator when you enter multiple PowerShell commands on one line. Alternatively, you can simply enter each PowerShell command on a separate line. As you can see in Figure 1-18, in the version I was using when I wrote this chapter, there were 129 cmdlets available to me. The figure you see may vary significantly, depending on whether additional cmdlets have been installed on your system.

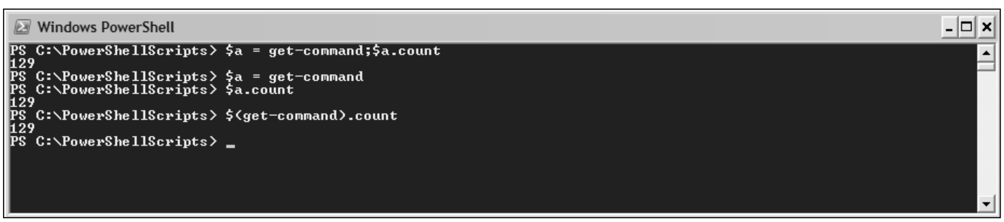


Figure 1-18

The first part of the command is an assignment statement:

```
$a = get-command
```

which assigns all the objects returned by the `get-command` cmdlet to the variable `$a`.

The second part of the command:

```
$a.count
```

uses the `count` property of the variable `$a` to return the number of cmdlets assigned earlier to `$a`. The default output is to the screen so the value of the `count` property is displayed on screen.

Scripting Language

PowerShell provides a new scripting language for the administration of Windows systems. Anything that you type at the command line can be stored as a PowerShell script and reused, as required, at a later date. Often, you will use PowerShell commands in an exploratory way on the command line to define and refine what you want to do. Once you have got things just right, you can store the commands in a PowerShell script and run the script at appropriate times.

In this example, you test a combination of commands on the command line with a view to saving them later as a simple script.

For example, suppose that you want to store in a text file the number of processes running on a machine together with date and time. You could do this by running the following commands, one at a time, on the command line:

1. First, assign to the variable `$a` the result of running the `get-process` cmdlet:

```
$a = get-process
```

2. Then assign to the variable `$b` the value returned from the `get-date` cmdlet:

```
$b = get-date
```

3. Then concatenate a label with the count of processes with the data and time converted to a string and assign the string to the variable `$c`:

```
$c = "Process Count: " + $a.count + " at " + $b.ToString()
```

4. To keep an eye on the current value of `$c`, write it to the host:

```
write-host $c
```

5. Then write the value of $c to a text file:

```
set-content C:\StoreCountAndDate.txt $c
```

6. After doing this, use the following command to show the current information in the text file:

```
get-content C:\StoreCountAndDate.txt
```

The result of this simple exploration is shown in Figure 1-19. The result displayed depends on how many times you have run the commands and at what times.

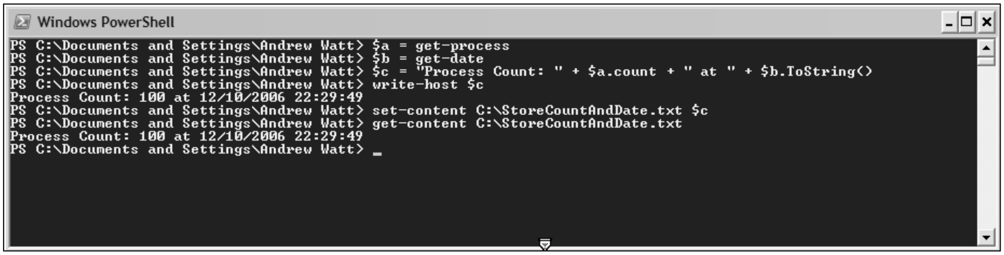


Figure 1-19

The `get-process` command returns all active processes on the machine.

The `get-date` cmdlet returns the current date and time.

You use the `count` property of the variable `$a` to return the number of processes that are active, then use string concatenation and assign that string to `$c`. The `ToString()` method of the `datetime` object converts the date and time to a string.

The `set-content` cmdlet adds information to the specified file. The `get-content` cmdlet retrieves the information contained in the specified text file. The default output is to the screen.

Once you have decided that the individual commands give the desired result — in this case, adding a count of active processes together with a date and time stamp to a selected text file — you can create a script to be run at appropriate times. To run the script, you need to enable script execution as described earlier in this chapter.

The following script, `StoreCountAndDate.ps1`, stores a count of active processes on a machine together with the current `datetime` value.

1. Open Notepad, or your other favorite editor, and type the following code:

```
$a = get-process
$b = get-date
$c = "Process Count: " + $a.count + " at " + $b.ToString()
write-host $c
set-content C:\StoreCountAndDate.txt $c
get-content C:\StoreCountAndDate.txt
```

2. Save the code in the current folder as `StoreCountAndDate.ps1`. The file extension for PowerShell version 1 scripts is `.ps1`. If you use Notepad, enclose the filename in quotation marks or Notepad will save the code as `StoreCountAndDate.ps1.txt`, which you won't be able to run as a PowerShell script.

3. Run the code by typing:

```
.\StoreCountAndDate
```

at the command line. This works even if the folder has not been added to the `PATH` environment variable.

The result should look similar to Figure 1-20.

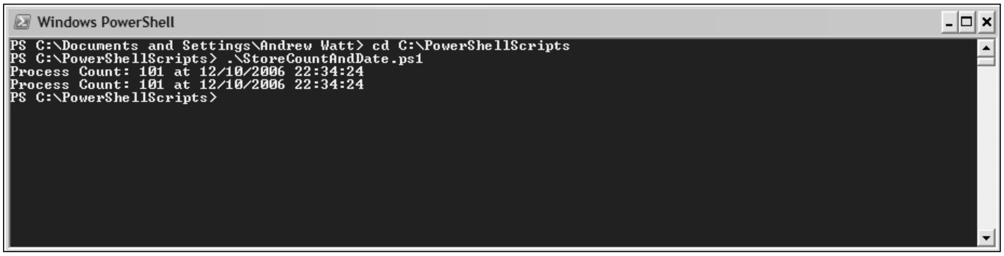


Figure 1-20

To run a PowerShell script in the current directory from the command line, type a period and a backslash followed by the name of the file (with or without the `ps1` suffix).

If your script is in the current working folder, just typing the script name does *not* work. This is because, with PowerShell, the current working folder (i.e., ".") is not part of the path. To run a script from the current folder, you have to explicity state the folder name. For example: `C:\PowerShellScripts\StoreCountAndDate.ps1` or `.\script.ps1`.

If you have difficulty running the script, it may be that running unsigned PowerShell scripts is not allowed on the computer you are using. If you follow the suggestion earlier in this chapter to set the execution policy to `RemoteSigned`, the script should run. Alternatively, you will need to sign the script in a way acceptable to your organization. I discuss signing scripts in Chapter 15

Summary

Windows PowerShell is a new command shell and scripting language for the Windows platform. This chapter showed you how to install the .NET Framework 2.0 and how to install Windows PowerShell.

In this chapter, you also learned how to carry out the following tasks:

- Start PowerShell
- Exit PowerShell
- Find out what PowerShell commands are available on your system
- Get help on individual PowerShell commands
- Develop and run a simple PowerShell script

You can, of course, create much more complex scripts in PowerShell than the example I showed in this chapter. Before going on, in Chapter 4, to begin to look in more detail at how some individual cmdlets can be used, I will step aside in Chapter 2 to look at the broader of issues of what is lacking in existing approaches and in Chapter 3 go on to look at the Windows PowerShell approach to improving on what was previously available.

2

The Need for Windows PowerShell

In this chapter, I briefly look at how Windows command line tools developed and some of the reasons why a new command shell and scripting language are needed on the Windows platform. In Chapter 3, I discuss some aspects of the approach that Windows PowerShell takes, with the aim of improving on the current command shell and scripting languages available on the Windows platform.

Windows PowerShell wasn't created in a vacuum. It has been created to fill a business need to allow administrators to work more effectively than the current command line and scripting tools on the Windows platform. Let's look at why Windows command line tools have been relatively neglected for years and why there was a business need for a better tool.

The world of computing is changing fast. In a business context, there is increasing pressure to get more work done faster and to do that work for the same or less cost. Twenty years ago, personal computers were just that — personal. It was good enough, in fact, it was pretty amazing at the time, to be able to process text (for example, in Word), business numbers (for example in Lotus 1-2-3), and data (for example, in dBASE) on a personal computer. The simple fact that one individual could work with information (whatever file format it happened to be stored in) was a huge step forward over the typewriter or adding machine (remember those?) that preceded the personal computer. In those days, only a small number of employees had a computer, and they tended to work alone, or if data was shared at all it was handed round on 5.25" floppy disks. Often information exchange would be on paper. Information from one program would be printed out on paper and read by a colleague. If that colleague needed to use that data in some program that he used, very often the data had to be entered into his program. Rekeying of data was an accepted evil in many businesses, simply because there was no practical way (other than at enormous cost) of moving data around between software packages.

Each user of a personal computer had, essentially, his or her own empire. They had autonomy (at least to some extent) about which programs were installed on their computer, how they configured the machine and its software to suit their personal way of working, and when, or if, software was updated. Since such users, typically, had no electronic data contact with other users in the same company or in other companies, it wasn't necessary to impose consistency about the configuration of the computer and its software. So tools, particularly command line tools, on DOS machines often focused on allowing a single user to carry out basic tasks appropriate to a single user using a single unconnected computer. In other words, the command line tools solved the problems that a single user needed to have solved. In the personal computer world at that time, many of the concepts that apply to networked computers, which we take for granted today, were unknown to most users.

Many users were almost hobbyists in their attitude, and many would dabble in command line tools and writing simple batch files, with each user often essentially being his or her own computer administrator. Nobody in those days had high expectations of usability from a personal computer, although they had improved usability compared to many larger predecessor systems. A personal computer was useful, but you had to fight it at times to get anything done. As time went on and increasing numbers of users wanted only to *use* a computer to get things done rather than spend time learning arcane (in their perception) commands and tweaking settings to get necessary tasks done, the usability limitations of command line tools became more obvious.

In the context of situations such as those I described in the preceding paragraphs, a move to a graphical user interface (GUI) had significant benefits for many users, since the interface was relatively simple and consistent to use. Microsoft seized a market opportunity, in part created by the difficulties many users found in mastering command line tools and in part created by the poor support from IBM for early versions of the OS/2 operating system. The sheer ease of use of early versions of Windows (despite its many limitations) created a rapidly expanding market opportunity for Microsoft, in both the operating system and application spaces. Of course, the move to event-driven programming also allowed users to work in a way that suited their circumstances or needs, which was simply impossible with the earlier DOS paradigm. In that context, a graphical user interface made a lot of sense (and it still does) for a single user. But as the number of computer users increased markedly and the networking of computers became more common, the issue of how to manage large number of machines has taken on increasing importance. In other words, graphical user interfaces had problems in scaling. For example, taking six clicks to carry out a task on one machine was fine. Six thousand clicks to do the task on one thousand machines was, and is, a problem.

As is well known, Microsoft made huge amounts of money from Windows and Windows-based applications. It was natural, therefore, that the company focussed on graphical-user-interface-based applications and tools. As a result Microsoft's command line tools have developed little from the DOS-based command line tools of a decade or two ago.

However, the world was moving on. Increasing numbers of personal computers were networked. Companies wanted to take increasing control of how individual computers, no longer so "personal," were configured. What had been genuinely a personal computer became more of a business machine. At first the advantages of standardized computing were perceived as affordable because software was changed infrequently. A gap of a few years between versions of software (at least those bought by a particular company) worked fairly well. It was expensive, but the economy in many Western countries made such an approach possible. But with changes in the global economy and national economies, there has been increasing pressure to reduce the costs of configuring, maintaining, and monitoring computers. It doesn't make any economic sense for a paid employee to travel around a work site manually configuring computers at frequent intervals. Of course, that sometimes tedious task isn't always avoidable, but it's economically a good thing to avoid if it's technically possible. Issues like these have provided a business case for Microsoft to improve its existing command line tools.

Today, as networked computers become the norm, it is increasingly important that all computers on a network can be managed by administrators without those administrators walking around office buildings or travelling between sites to do so. And, where appropriate, the administrators should be able remotely to find out the state and modify the configuration of those machines to conform to some enterprise standard or be updated in a controlled and tested way. With the command lines tools before PowerShell, administrators were very limited in what they could do to manage Windows machines, at least with the tools that were part of the Windows distributions or were free. Until PowerShell, at least in the Windows world, effective command line support for administrators tended to slip between the cracks.

Limitations of CMD.exe

The traditional Windows command shell hasn't changed fundamentally since the days of DOS, although as time has passed some new commands have been added to it. `CMD.exe` allows a user or administrator to carry out simple tasks such as listing the files in a directory using the `dir` command or format a disk using the `format` command, but it certainly doesn't provide anything remotely like a comprehensive tool to administer a single Windows machine. Many tasks that you want to carry out on your machine can only be done using the Windows graphical user interface. In fact, it's not one graphical user interface that you need to master. You need to use several tools to get a job done.

If it's a task that one user does once or only occasionally, then the GUI tools save the user the time it would take to learn the details of multiple command line commands. However, when it comes to trying to administer dozens, hundreds, or thousands of machines, then `CMD.exe` simply doesn't even come close to having what it takes to get the job done. Admittedly, some commands, such as `AT`, allow you to run a command on a remote computer, so you're not totally confined to administering a single machine. But the coverage an administrator needs is far from adequate with `CMD.exe`.

> **The Windows NT (and later) command line utility, `CMD.exe`, replaced the DOS and Windows 9X `Command.com`. Visually and functionally the changes were minor, although over time a significant number of commands were added. In practice, neither utility allowed a user or administrator to carry out anything other than relatively minor tasks. `CMD.exe`, like its DOS predecessor, was designed largely in the context of a single machine rather than a network of large numbers of interconnected machines.**

The relative poverty of functionality in `CMD.exe` isn't too surprising. Microsoft's focus was elsewhere in developing GUI tools. One problem that Windows was intended to solve was the need for users to remember huge numbers of potentially unfriendly switches and arguments that DOS commands needed. If the aim of Windows and its GUI tools is to avoid users having to learn command line commands, then why provide tools that require learning what you're trying to help users avoid? For users managing a single machine (if they actively manage any machine at all), a graphical user interface's consistency and relative simplicity of interaction is a potentially significant step forward for many users. However, when you need to manage hundreds or thousands of machines, a graphical tool becomes a tedious bore, with click following repetitive click. For businesses with large numbers of computers, such an approach is not only inefficient but expensive.

Ease of administration of multiple machines is likely to have been one of several factors in why Linux and similar operating systems have begun to eat into Microsoft's markets, not the least in the server sector. In that context, PowerShell can be seen a defensive move by Microsoft to provide a flavor of Windows that attempts to take back the administrative high ground.

If you need convincing of the limitations of `CMD.exe`, take a look at the commands that are available. To view all available commands in the existing Microsoft command line shell, simply type `Help` at the command line, and all the commands will be listed, together with a brief description of what each command does. But that is part of the problem. The help available isn't easy to read nor is it comprehensive. Realistically, in the context of the Windows emphasis on the graphical user interface, the command line way of working has been very much a second class citizen.

In addition, the toolset of the existing Windows command shell has several significant limitations.

Batch Files

When you are able to do what you need from the command line you can capture the commands in a batch (`.bat`) file, and that's great — as far as it goes. If you're not writing batch files regularly, then you may well find that you can't remember the exact syntax you need to create logic that satisfies anything but the simplest needs. The language used in batch files is pretty archaic, and when it was created user-friendliness wasn't a high priority. Maintenance of batch files can be tedious, too, particularly if they are long and were written by someone else.

Yes, batch files can work. But their support for `IF` and `GOTO` seems to belong to another era, as in fact it does. But if you are using a batch file and find you can't easily stretch it to do something a little more complex than `IF` and `GOTO` will support, what do you do next? There is no easy step up from the syntax for batch files. In other words, it's a syntax dead end. Switching to a scripting language like VBScript or JScript means that you need to learn (or relearn) a scripting language with a very different syntax from batch files. You also need some familiarity with the underlying object structure that the scripting language is going to access or manipulate.

If a scripting language doesn't give you the performance or functionality that you want, then you have another step up to make, perhaps to Visual Basic (pre- or post-.NET) or C#. Either way, there are significant further changes in syntax.

Inconsistency of Implementation

Another issue in using command line tools was that they were created by different teams at Microsoft. Those teams worked, to a significant extent, in isolation, like the users of a decade or so before, and that resulted in a lack of consistency in how commands were implemented in different command line tools. In individual tools, the syntax to use parameters in one tool would differ from the parameter syntax in another tool. Such inconsistencies add to the learning curve for those tools. Since Microsoft's focus was on GUI tools in Windows, there was no high-level push to standardize command line tools.

Inability to Answer Questions

There are a huge number of tasks that `CMD.exe` is incapable of performing. For example, you could not discover from the command line interface (before Windows Management Instrumentation) which processes were running on a machine or which services were currently running.

The gaps are so huge that it's simplest and most honest just to say that they are there and that they're huge. `CMD.exe` is simply not, in my opinion, a tool fit for comprehensively administering one Windows machine, never mind large numbers of them.

Lack of Integration with GUI Tools

In all versions of Windows, GUI tools have been a major way to carry out administrative tasks across a wide range of Microsoft and third-party products intended to run on the Windows platform. Using a GUI to administer one machine can be relatively fast and effective. But, if you have to carry out the same sequence of clicks on 5, 10, 100 or 1,000 machines, the limitations in scalability of a GUI-based approach becomes very clear and, as numbers of machines increase, very inefficient and frustrating.

GUI tools often had no easy mapping to the available command line tools. So, for some tasks you had the opportunity to use command line tools, but for others the only option was to use a GUI tool. There was no easy way to find out if something you could do with a GUI tool could also be done from the command line. One result of that was that carrying out a task on a single machine using a GUI didn't help you at all with carrying out the same task subsequently on multiple machines.

The GUI Emphasis in Windows

One of the guiding principles when Microsoft moved from the character-based DOS operating system to Windows was that graphical user interfaces provided ways to carry out tasks that were much more convenient than when using DOS-based command line tools. Users had problems finding, understanding, or remembering command line commands and their switches and parameters. The GUI metaphor worked better than the command line, at least for those users who were unable (or unwilling) to master the syntax of command line tools.

For many tasks, the GUI-based approach undoubtedly works well. For other tasks, particularly system administration tasks, GUI tools can be productive when used on a small scale but become extremely tedious to use when the same task has to be carried out on a dozen, a hundred, or a thousand machines.

Previous Attempted Solutions

Microsoft has made several previous attempts to address the kinds of issues mentioned earlier in this chapter. Each attempt has, not surprisingly, taken some steps forward, but each has had limitations. Not least of the limitations for the Windows user in an increasingly .NET Framework–orientated world is that the existing technologies don't use the .NET Framework nor do they generate or execute managed code.

Windows Script Host

Windows Script Host (WSH) was introduced in 1998. One important aim of Windows Script Host was to enable various scripting languages to support a range of Windows administration tasks.

Windows Script Host didn't prove to be popular. One reason, I suspect, was that documentation of how best to use Windows Script Host wasn't easy to find in the early years of its life. Naturally, administrators were reluctant to use a tool that they couldn't easily locate information for.

Another factor in the relatively poor uptake of Windows Script Host was the occurrence of several security exploits. Of course, WSH was by no means the only Microsoft product that exhibited worrying security vulnerabilities, but a questionable reputation for security isn't an encouragement to the rapid uptake of a scripting environment.

Having made those negative comments, it's fair to say that WSH allows the scripter who uses VBScript, JScript, or other scripting language to carry out many useful administration tasks. One of the major parts of the learning curve for administrators was the need to learn about the Component Object Model (COM). For many Windows developers, that model was almost second nature. For administrators, it was typically unfamiliar territory. The result was that many administrators lacked the time or motivation to develop sufficient knowledge of COM APIs to be able to effectively and efficiently carry out routine administrative tasks. An administrative tool that required less knowledge of the underlying application programming interfaces (APIs) to get started would be an improvement.

One unavoidable disadvantage of WSH is that the languages you use in WSH scripts are not the languages you use on the command line or the batch language used to automate command line commands. So, if you like to explore a machine interactively and find the commands you want to carry out a specific task, you can't simply go on and use those same commands in your VBScript or JScript code. As you will see in later chapters, Windows PowerShell provides a better path from the command line to scripts.

For the sake of efficiency, a tool is needed to allow administration of one or many machines from the command line. Applying a script with identical commands across dozens or hundreds of machines is more consistent and more time-efficient than using a GUI to administer large numbers of machines.

Windows Management Instrumentation

Windows Management Instrumentation (WMI) addresses some of the issues that PowerShell attempts to address. Microsoft was involved in the creating the context to WMI — WBEM (Web-based enterprise management). The WBEM initiative was picked up by the Distributed Management Task Force (DMTF) to produce a cross-platform standard for management in a distributed, enterprise computing environment.

WMI tools aren't particularly user friendly to the uninitiated. The Windows Management Instrumentation Command-line tool (WMIC), operates via aliases, which attempt to abstract away the need for detailed knowledge of WMI classes. But WMIC is itself less than user-friendly to the new user. Other WMI tools need to be downloaded separately and demand some knowledge of WMI architecture to use them even for simple tasks.

WMI provided a more consistent interface than programming directly against the COM model allowed. However, WMI has a huge number of classes and properties to master. In addition, writing WMI scripts can be lengthy and tedious. For example, if you want to display multiple property values, it becomes really tedious to write multiple times `vbcrlf` in your code plus line continuation characters and so on. It gets the job done, but I don't find it an enjoyable process.

In the context of Microsoft's current, and likely future, emphasis on code using the .NET Framework, the fact that neither VBScript nor WMI are .NET-based is a significant factor against WMI going forward. Of course, WMI scripts will continue to work, but it seems likely to me that no substantive further development of WMI will take place. WMI is very useful, but probably WMI will now be a very useful dead end. WMI isn't going away any time soon though. Part of WMI's ongoing usefulness is the ability of PowerShell's `get-wmiobject` to retrieve information using WMI classes. So, if you've invested time in learning about WMI classes, that knowledge will prove useful when using PowerShell. I discuss using WMI from PowerShell in Chapter 23.

Summary

Windows command line tools have existed for many years in a context where graphical tools were Microsoft's preferred approach. As a result, development of the Windows command line has been neglected and doesn't meet the needs of today's businesses or system administrators.

- The toolset of `CMD.exe` covers only a limited range of the tasks that an administrator needs to carry out.
- The syntax of command line tools and batch files means that batch files are limited in the logic they can easily implement. Lengthy batch files are often difficult to read and understand.
- The language of batch files is very different from the scripting languages, such as VBScript and JScript, needed to get more complex tasks done.
- There is no way to capture a task done using a graphical tool and create the corresponding command line syntax.

In Chapter 3, I will describe the approach that Windows PowerShell takes to these issues and discuss how, even in version 1, Windows PowerShell provides a better and more consistent way to handle a substantial number of system administration tasks.

3

The Windows PowerShell Approach

The PowerShell team recognized many limitations of the existing Microsoft command line, GUI, and scripting tools which I described in Chapter 2. The background against which the PowerShell team was working was changing significantly with a strategic move at Microsoft from COM (Component Object Model) programming to .NET Framework programming. It therefore made sense, going forward, for PowerShell to be based on the .NET Framework.

The move from COM-based programming to .NET Framework–based programming opened up opportunities to create a new approach to the command line and a new scripting language using the same commands and syntax as were available on the command line.

A New Architecture

PowerShell 1.0 implements a significant new architecture, different from any preceding Microsoft command shell. First, it is based on the .NET Framework version 2.0. Second, instead of the traditional approach of command shell pipelines, which often pass strings or text from one application to another, the PowerShell approach is to pass objects, rather than text, along the pipeline. In PowerShell the objects are .NET objects.

.NET Framework-Based Architecture

It probably bears repeating that one of the most significant changes in PowerShell as a command shell is that it is based on the .NET Framework 2.0. Among the relevant features of the .NET Framework are

- ❑ Reflection
- ❑ Network awareness
- ❑ Rapid application development

In the context of PowerShell, reflection is particularly important. You can find the members of any .NET class at runtime using the `get-member` cmdlet. For example, to find the members of running processes that you can work with, use the following command:

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

The `get-process` cmdlet, in the first step of the pipeline, returns `System.Diagnostics.Process` objects and passes these to the next step in the PowerShell pipeline. In the second pipeline step, the `get-member` cmdlet returns objects representing the members of the `System.Diagnostics.Process` class. This process is enabled through .NET reflection.

Windows PowerShell provides a syntax that allows users to make use of static members any .NET Framework 2.0 class. For example, to find the current time using that syntax and assign it to a variable `$now`, type the following command in the PowerShell console:

```
$now = [System.DateTime]::Now
```

As you can see, to call a .NET class, you enclose the class name in paired square brackets, and provide the method or property name separated by two colons. In this case, the command uses the `Now` static property of the `System.DateTime` class and assigns that to the variable `$now`. The same technique can be used to employ the methods of any .NET Framework 2.0 class or get or set the value of any property of a .NET class. Alternatively, you can use the `get_now()static` method of the `System.DateTime` class to achieve the same result:

```
$now = [System.DateTime]::get_now()
```

To display the value of the variable `$now`, simply type the following command:

```
$now
```

The current date and time are displayed on the PowerShell console. In traditional command shells, after you have assigned a date and time to a variable, you have to use string parsing to find desired components of the date and time. Issues such as whether the date is in the MM/DD/YYYY, DD/MM/YYYY, or YYYY/MM/DD format also come into play if the date is held as a string. However, PowerShell offers advantages in this respect, too. For example, you can unambiguously access the month of a `System.DateTime` object, using the `month` property of a `DateTime` object:

```
$now = [System.DateTime]::get_now()
$now.month
```

To find the members of a .NET Framework object, use the `get-member` cmdlet.

```
[System.DateTime] |
get-member
```

You can specify that only methods be displayed, using

```
[System.DateTime] |
get-member -memberType method
```

or only properties, using

```
[System.DateTime] |
get-member -memberType property
```

To display the static members of a .NET Framework class, use the `-static` parameter:

```
[System.DateTime] |
get-member -static
```

or to display only static methods or properties combine the `-static` parameter with the `-memberType` parameter. For example, to display static properties of the `System.DateTime` class use the following command:

```
[System.DateTime] |
get-member -static -memberType property
```

A useful source of information about .NET classes is in the documentation that forms part of the .NET Framework 2.0 SDK. This can be downloaded from `http://msdn.microsoft.com/netframework/downloads/updates/default.aspx`*. At the time of writing versions of the SDK are available for x86, x64, and IA64.*

Object-Based Architecture

Windows PowerShell, because it is based on the .NET Framework, is object-based. Many other command shells are, in essence, pipelines of text. The fact that objects are passed along a pipeline has several advantages.

For example, when using objects in a pipeline, you no longer have to use string parsing to retrieve desired components of the date. Assume that you have created a PowerShell variable as follows:

```
$now = [System.DateTime]::get_now()
```

To retrieve the year, simply type

```
$now.Year
```

and the value of the `Year` property of the `$now` variable is displayed. Similarly, the command

```
$now.DayOfWeek
```

displays the day of the week for the current date. Figure 3-1 shows the result of running the preceding commands.

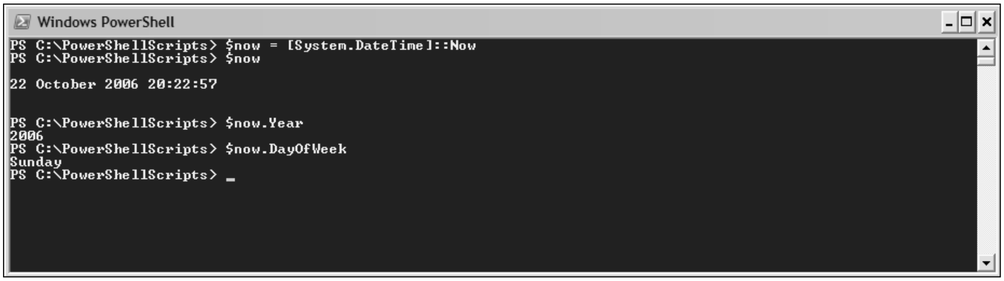


Figure 3-1

If you're not familiar with the members of a particular .NET class, use the PowerShell `get-member` cmdlet to display the members of an instance of the .NET class. For example, if you wanted to view information about the methods of the variable `$now`, which is a `System.DateTime` object, you could use the following command, given the assignment of a `DateTime` object to `$now`:

```
$now |
get-member |
more
```

The `get-member` cmdlet used without parameters displays all members of a .NET class. The output is piped to the `more` alias to display only one screen of information at a time, as shown in Figure 3-2. By the way, if you need to be convinced that `$now` is an object of `System.DateTime` take note of the first information displayed in Figure 3-2.

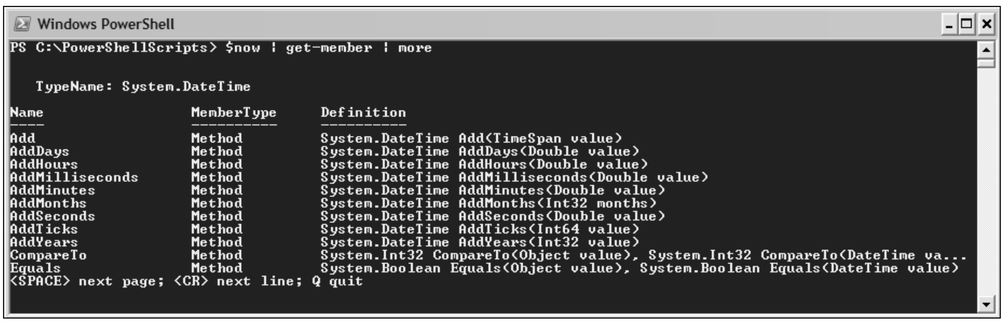


Figure 3-2

PowerShell allows you to access .NET Framework Class Library functionality, but for some tasks you don't need to take route. For example, finding the current date and time isn't something you need to use the preceding syntax for, since PowerShell has a cmdlet to do that, called `get-date`. Execute the following commands to assign the current date and time to the variable `$now`:

```
$now = get-date
$now
$now.GetType()
$now.GetType().Fullname
```

As you can see in Figure 3-3, the `get-date` cmdlet produces a .NET object (of type `System.DateTime`), which can be manipulated as shown earlier.

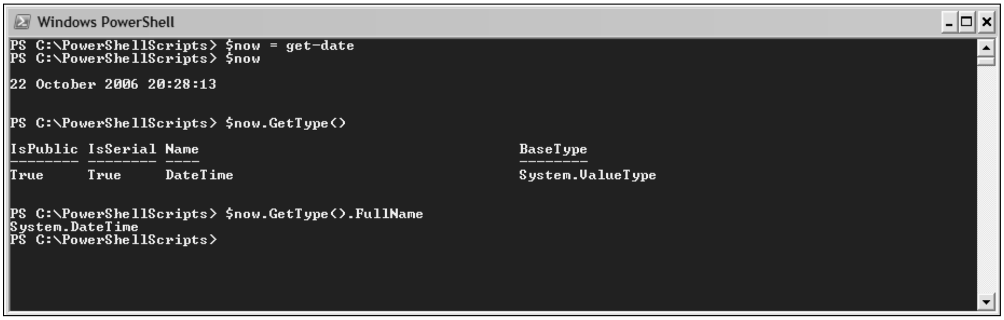


Figure 3-3

```
$now = get-date
```

uses the `get-date` cmdlet to assign the current date and time to the variable `$now`. You can display that value using the command:

```
$now
```

To display the type of `$now`, use the `GetType()` method:

```
$now.GetType()
```

If you are unsure what namespace the `DateTime` class belongs to,

```
$now.GetType().Fullname
```

returns the value `System.DateTime`, which is the full name of the class.

The .NET Framework class library is huge. PowerShell's ability to use the .NET Framework class library allows it to reach into very many places in a Windows installation. Everywhere that a .NET class has methods or properties you can use PowerShell to exploit the power of those .NET classes from the command line or by using PowerShell scripts.

In version 1.0 of PowerShell, the number of cmdlets is fairly limited — at least a lot of specialized cmdlet functionality will be available separately with products such as Exchange Server 2007. Powershell version 1.0 has 129 cmdlets. To confirm how many cmdlets are available to you in the PowerShell build that you are using, use the following command.

```
(get-command * -CommandType Cmdlet).count
```

The number of cmdlets available is displayed on the command line, as shown in Figure 3-4.

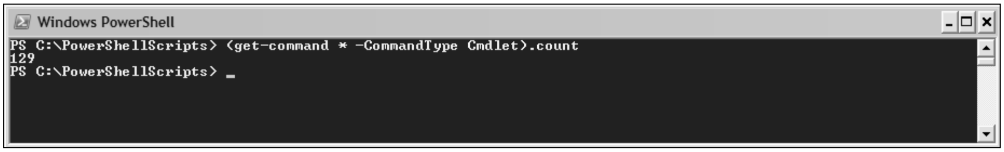


Figure 3-4

If you have other products installed which use PowerShell cmdlets under the covers (for example Exchange Server 2007), the number of cmdlets displayed after running the preceding command may be much larger.

The cmdlets included in Exchange Server 2007 are not covered in this book. At the time of writing, it is likely that about 350 cmdlets specific to management of the Exchange Server will be made available in Exchange 2007.

A New Cross-Tool Approach

PowerShell is intended to provide nearly complete coverage of the administration tasks of Windows machines using cmdlets. Not everything that is needed to do that is available in PowerShell version 1. One approach you can use to fill in the gaps in PowerShell 1.0 is to use .NET classes, as described in the preceding section. Another approach is to use Windows Management Instrumentation. PowerShell provides a `get-wmiobject` cmdlet to allow you to retrieve information about machine state. For example, to retrieve the current date and time using WMI, type the following command:

```
get-wmiobject -Namespace root\cimv2 -Class Win32_CurrentTime
```

which can be abbreviated to:

```
get-wmiobject Win32_CurrentTime
```

As you can see in Figure 3-5, information about the current date and time is displayed.

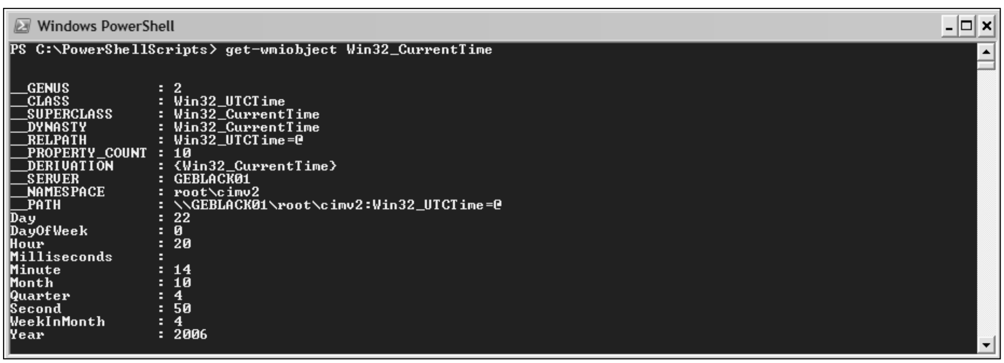


Figure 3-5

One use of WMI that is particularly important in PowerShell version 1.0 is accessing remote machines. The core cmdlets in PowerShell 1.0 only access the local machine.

An alternative approach to remote machine access using cmdlets is to use .NET Framework classes. However, some Exchange Server 2007 cmdlets have support for accessing remote machines.

GUI Shell (MMC Layered over PowerShell)

The aim of the Windows PowerShell team is that the next generation of the Microsoft Management Console, MMC 3.0, will provide a graphical user interface (GUI) layered over PowerShell commands. It seems likely that several next-generation Microsoft products will have PowerShell functionality as the basis for their management tools. This dual functionality will first be delivered in Exchange Server 2007.

In Exchange Server 2007 the next-generation MMC tools generate PowerShell scripts from GUI actions, in much the same way that you can currently generate T-SQL scripts from the graphical SQL Server 2005 Management Studio interface. The scripts you create from the MMC 3.0 GUI can, of course, be adapted for example to carry out the same actions for all machines in a desired collection. So, it is likely that you will be able to use GUI skills to create PowerShell scripts or at least to create PowerShell script templates that you can adapt or incorporate into more sophisticated scripts.

Command Line

Often when you are trying to figure out how best to use PowerShell to solve a problem, you will initially work in the shell on the command line in an exploratory way. This allows you to quickly observe the actual results you get from executing a PowerShell command and, for example, modify the value of one or more cmdlet parameters to tweak the behavior of the command (or pipeline of commands) to achieve just what you want.

Often when applying PowerShell from the command line in an exploratory way, it makes good sense to use the `-whatif` switch. Doing so allows you to see what *would* have happened if you had executed the command, before PowerShell actually changes anything on the system. This is much more sensible than diving in and possibly damaging a system. Suppose that you want to delete some files. You might think that you know exactly what you want to do. For example, if there were several files you wanted to delete from the `Pro PowerShell\Chapter 03` directory, you could use a command like this to delete all files beginning with t which are `.txt` files:

```
remove-item "C:\Pro PowerShell\Chapter 03\t*.txt"
```

This could result in PowerShell deleting files that you may not have intended to delete. It is safer to run the command first with the `whatif` parameter specified, as follows:

```
remove-item "C:\Pro PowerShell\Chapter 03\t*.txt" -whatif
```

Figure 3-6 shows the kind of message you will receive if you specify the `whatif` parameter. The message tells you what PowerShell would have done if you hadn't specified the `whatif` parameter. Nothing has been deleted. If the files to be deleted are the ones you want to delete, simply remove the `whatif` parameter and run the command again to actually delete the files.

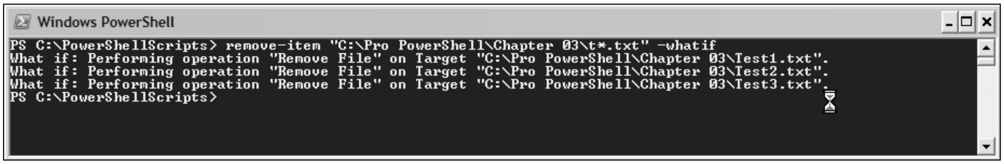


Figure 3-6

If you want to try this out yourself, create some files in the `Chapter 03` directory, for example by using the following commands:

```
"test" > "C:\Pro PowerShell\Chapter 03\Test1.txt"
"test" > "C:\Pro PowerShell\Chapter 03\Test2.txt"
"test" > "C:\Pro PowerShell\Chapter 03\Test3.txt"
```

The redirection operator, `>`, sends the test string to a file as named in the path, which forms the latter part of each command.

You can run all the core cmdlets from PowerShell's command line in this way. Typically, you will create pipelines of cmdlets with objects created in one step of the pipeline passed to the next step of the pipeline for some further processing. You may, for example, use the `where-object` cmdlet to filter objects passed to that step by an earlier step.

The following command creates a pipeline that looks for `.dll` files in the `C:\Windows\System32` folder, selects `FileInfo` objects where the DLL was created in 2006, sorts those in ascending date order, and displays a two-column table containing the name of the DLL and the date and time when it was created. For ease of reading, I have put each step of the pipeline on its own line on the page. The command assumes that you installed Windows in the `C:\Windows` folder.

```
get-childitem -Path C:\Windows\System32 -Filter *.dll |
where-object {$_.CreationTime.Year -eq "2006"} |
sort-object CreationTime |
format-table Name, CreationTime
```

Figure 3-7 shows the part of the results generated by the preceding command.

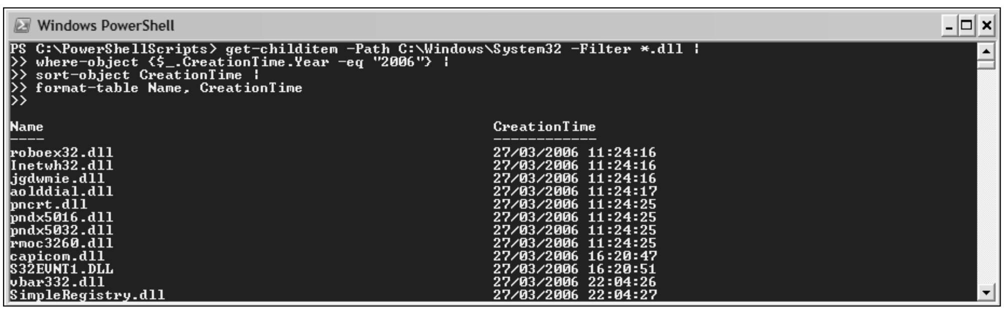


Figure 3-7

In the first step of the pipeline, the `get-childitem` cmdlet finds child items in a specified folder. In a folder, child items are either files or other folders. The `Filter` parameter specifies which child items are to be selected (i.e., `.dll` files).

Next, the `where-object` cmdlet in the second step of the pipeline filters the objects passed to it, and the `sort-object` cmdlet sorts the filtered objects in ascending order by creation time. The final step uses the `format-table` cmdlet to produce a two-column table for display.

Command Scripting

Once you are satisfied that you have the right output or effects, you can include PowerShell command lines in a PowerShell script file.

The PowerShell command line doesn't make it really convenient to copy commands into a text editor. One simple technique is to clear the screen then run each of the commands (which can be accessed by using the up and down arrows) that you want to incorporate in the script, choose Edit ➪ Select All in the command shell window's menu, then press Return to copy all the selected text. You can then paste that text into a text editor and delete the prompts that you copied from the screen. Alternatively, you can drag across desired text (you can only select a rectangular block) and right-click to copy it.

PowerShell includes the `start-transcript` and `stop-transcript` cmdlets. The `start-transcript` cmdlet redirects a copy of everything that is typed on the command line and displayed on the screen to a file. You can then open the transcript file after completion of a PowerShell session and copy and paste desired commands from the transcript to your selected text or code editor. This is better than using session history, since it captures all commands in a session (unlike the session history, which stores a specified maximum number commands) and also permanently stores them in a file (which session history does not). Depending on how you work with PowerShell, you may want to issue `start-transcript` as the first command of a PowerShell session. Alternatively, and more conveniently, add the `start-transcript` command to a profile file that PowerShell will load before you type your first command. The parameters of the `start-transcript` cmdlet allow you to send the output to any selected directory.

Other options are becoming available at the time of writing. For example, Karl Prosser's PowerShell Analyzer allows you to enter PowerShell commands in a text editor pane and view the results in a pane that looks similar to the PowerShell shell. Figure 3-8 shows an early build of PowerShell Analyzer. You can find further information about PowerShell Analyzer at `www.powershellanalyzer.com` and download it from there.

Figure 3-9 shows the situation when typing the `get-date` cmdlet and shows the IntelliSense-like support in the editor. Notice, too, that the version I was testing didn't echo the PowerShell command in the shell. This is a much more convenient environment in which to develop scripts. You can try out a command or series of commands and tune the results to what you want. Once you have created the desired functionality using a cmdlet or a pipeline simply select File ➪ Save As to save the script in a desired location.

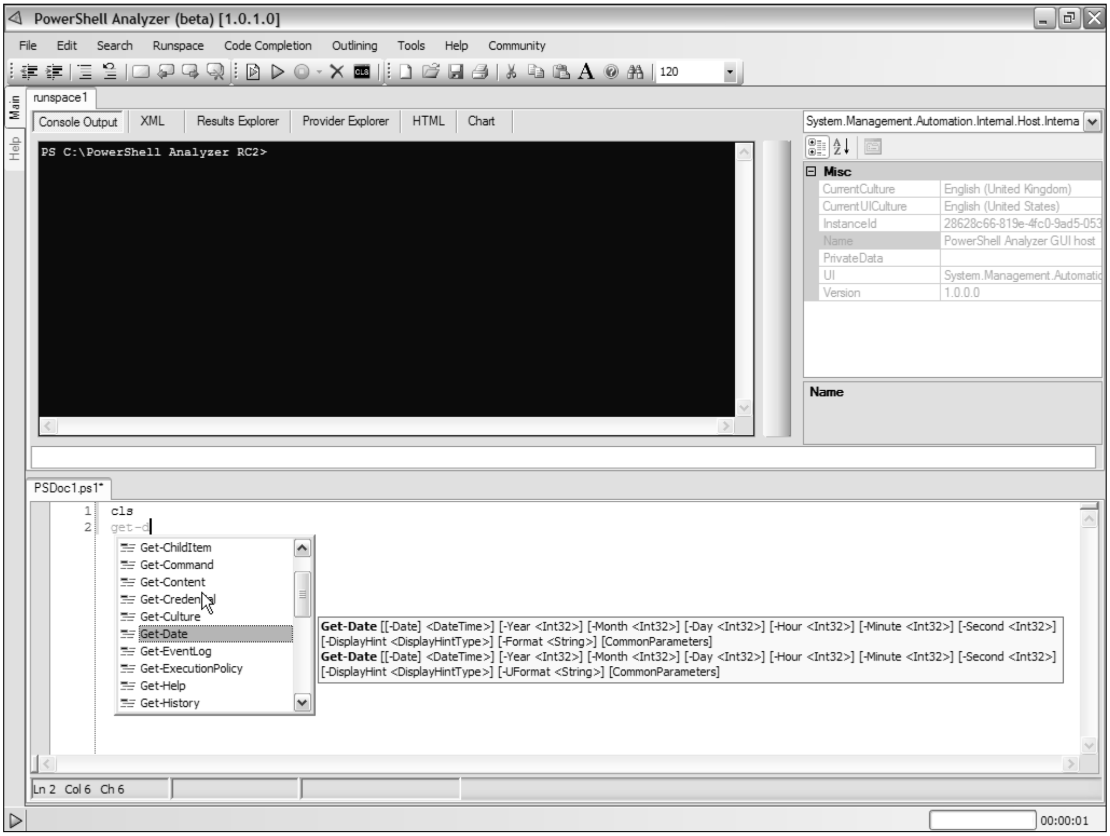


Figure 3-8

Another new tool available in development at the time of writing is PowerShell IDE from Tobias Weltner. Like PowerShell Analyzer it offers a code editor pane and a shell-like pane among other features. When you run a PowerShell command, or series of commands, they are echoed in the shell.

For further up-to-date information on PowerShell IDE, visit `www.powershell.com`.

Commercial scripting tools will also offer support for development of PowerShell scripts. At the time of writing beta builds of the well-respected PrimalScript editor are available with support for Windows PowerShell. For further information about PrimalScript visit www.sapien.com.

I describe writing PowerShell scripts in more detail in Chapter 10.

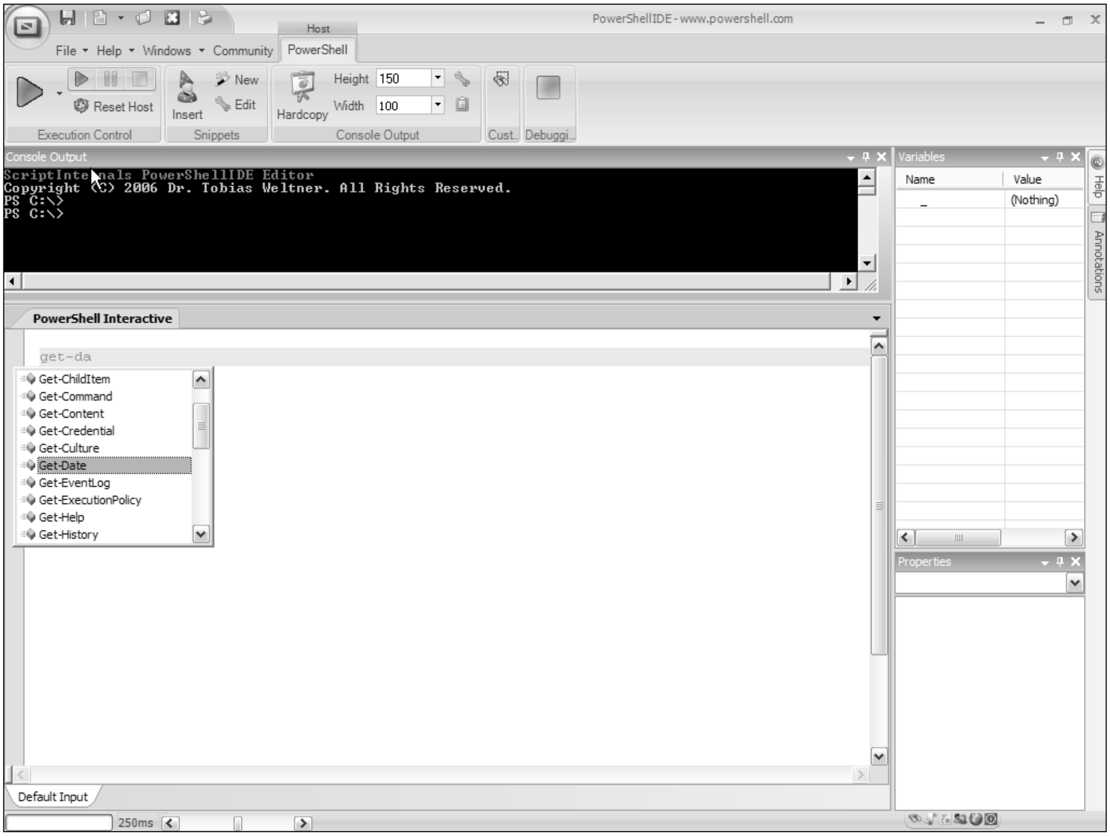


Figure 3-9

COM Scripting

Although PowerShell is based on the .NET Framework, you can also carry out scripting of COM objects. This has a couple of advantages. You can leverage any existing knowledge you have of how to manipulate COM objects. It also fills in gaps that the cmdlets don't cover in version 1.0 of PowerShell.

The `new-object` cmdlet, when used with the `ComObject` parameter, allows you to create a new COM object. You can then manipulate that COM object, as you need to.

To create an instance of Internet Explorer from the command line, use the following command:

```
$ie = new-object –ComObject InternetExplorer.Application
```

The COM object is assigned to the variable `$ie`. This appears to do nothing since, by default, a newly created instance of Internet Explorer is not visible. However, if you execute the following command, which makes the newly created Internet Explorer instance visible, you can then see that you have automated an instance of the browser.

```
$ie.visible = $true
```

Notice that PowerShell represents true as `$true` (and false as `$false`).

You can then use the methods and properties of the object you created to automate Internet Explorer. For example, you can navigate to a specified URL:

```
$ie.navigate2("http://andrwwatt.wordpress.com")
```

This is shown in Figure 3-10.

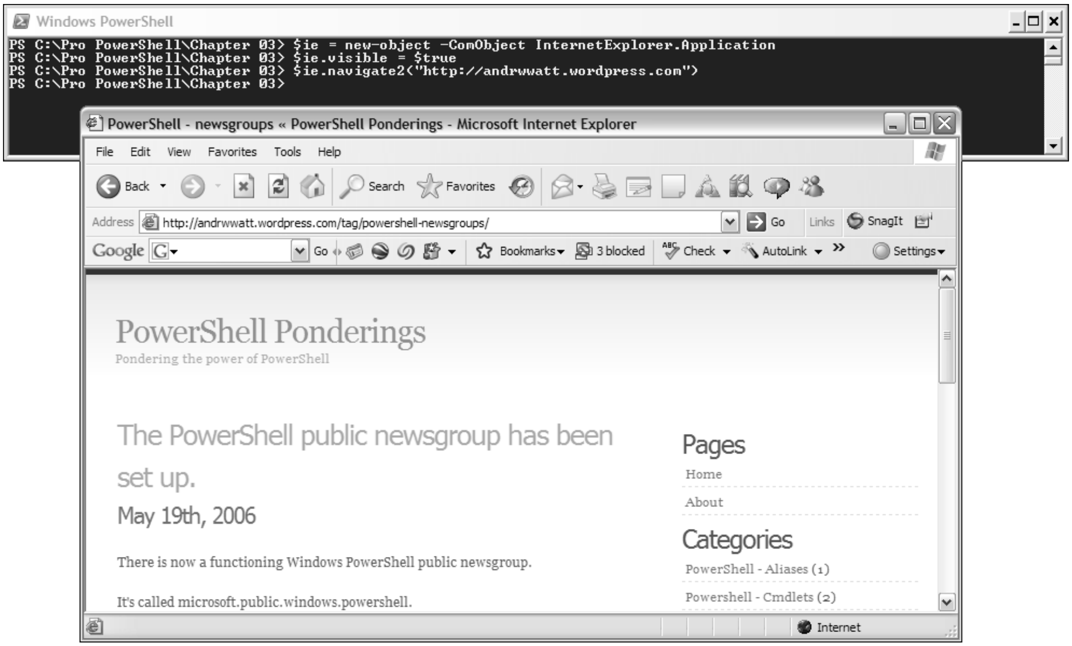


Figure 3-10

I describe COM scripting in more detail in Chapter 13.

Namespaces as Drives

In PowerShell several data stores are exposed as drives. When you work with files and folders, you expect to see a drive as the container. In Windows PowerShell the registry, aliases, certificates, environment variables, functions, and variables are all exposed to you as drives. In other words, you can use the same cmdlets to work on items in the file system and the registry, aliases, certificates, functions, and environment variables.

Each of the data stores exposed by PowerShell as a drive is underpinned by a *command shell provider*. A command shell provider maps underlying data structures so that you can work with the data as if it were stored in folders and files. To display the providers available on your system, use this command:

```
get-psdrive |
group-object Provider |
format-list Name, Count
```

Figure 3-11 shows the providers available on one Windows XP system. Notice that the `FileSystem` provider supports several drives.

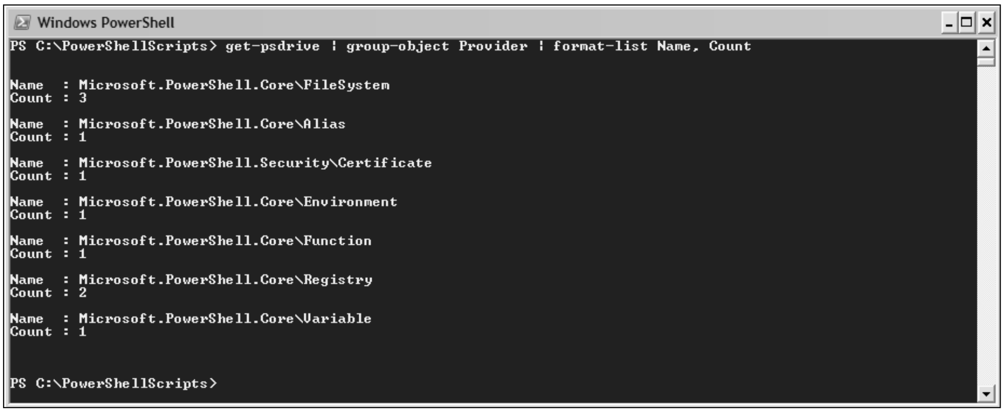


Figure 3-11

The `get-psdrive` cmdlet retrieves all drives defined on your system. In the second step of the pipeline, the `group-object` cmdlet groups objects passed to it by the first step of the pipeline according to the command shell provider. The final step of the pipeline formats the groups as a list, with the name of each group and the count in each displayed.

An alternative approach is to use the get-psprovider cmdlet:

```
get-psprovider
```

which produces results similar to those shown in Figure 3-12.

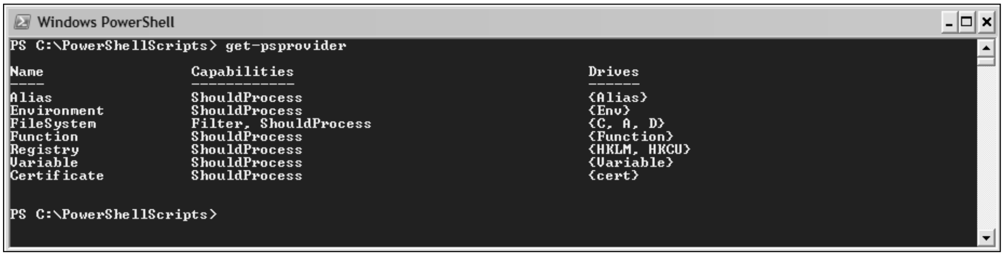


Figure 3-12

Notice that this approach also displays information about the capabilities of the available providers. On this machine there are three drives that use the `FileSystem` provider. Notice, too, that there are two drives, HKLM and HKCU, that use the `Registry` provider. The HKLM drive corresponds to the `HKEY_LOCAL_MACHINE` hive in the registry. The HKCU drive corresponds to the `HKEY_CURRENT_USER` hive.

Since the registry is exposed as a drive, you can navigate to it just as you can to a file system drive. For example, to move to the HKLM drive, simply type:

```
set-location HKLM:
```

The `set-location` cmdlet is used to set a new location. More conveniently, you can use the `cd` command as this is an alias for `set-location`:

```
cd HKLM:
```

Whether you use the `set-location` cmdlet explicitly or by using the `cd` alias, by default, the prompt changes to indicate the new current working directory. To display the content of the HKLM drive, use this command:

```
get-childitem *
```

The * character in the preceding command is a wildcard, which matches all child items. As you can see in Figure 3-13, there are four children. Access is denied to the SAM child item. Similarly, if you use the Regedit utility, you cannot see the content of SAM.

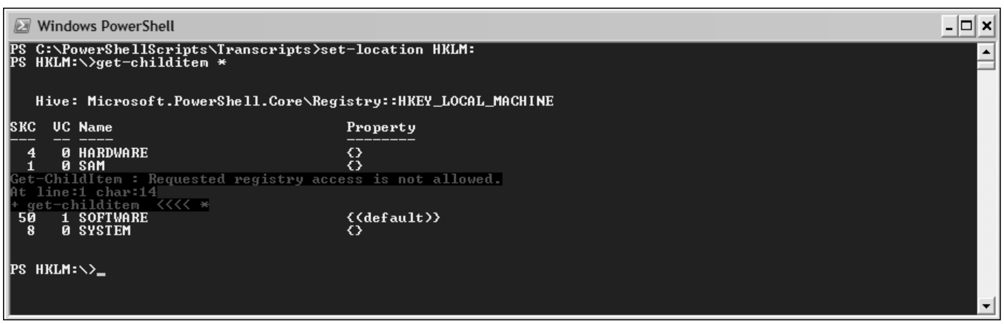


Figure 3-13

The `$pwd` variable contains information about the current working directory. To display the current working directory, simply type:

```
$pwd
```

The current location in the currently used command shell provider is displayed. If you followed the preceding commands the result `HKLM:\` is displayed.

File System Provider

The `FileSystem` provider allows you to work with drives, folders, and files in ways similar to the familiar techniques you use in `CMD.exe`. There are specific cmdlets to retrieve information about drives (`get-psdrive`) and folders and files (`get-childitem`). To ease the transition toward using these cmdlets, you can, by using built-in aliases, apply familiar commands like `dir` to find the child items in a folder. To do that without using an alias, you use the `get-childitem` cmdlet.

The following commands use the `dir` alias to retrieve `.txt` files in the `C:\Pro PowerShell\Chapter 03` folder (assuming that it is the current directory).

```
dir *.txt
```

The equivalent command using the `get-childitem` cmdlet is:

```
get-childitem *.txt
```

The `dir` alias uses the `get-childitem` cmdlet under the covers. There is no significant performance benefit either way. It's simply a matter of convenience or preference.

Registry

The `Registry` provider is a command shell provider that allows you to work with registry keys and values in ways similar to those you use to work with files and folders. For example, to move to the `HKEY_CURRENT USER` hive in the registry and find its child items, use the following commands:

```
set-location HKCU:
get-childitem *
```

The `set-location` cmdlet sets the current working directory (which is contained in the `$pwd` variable). The colon must be included after the drive name. The `get-childitem` cmdlet retrieves information about the hives in the HKCU drive. Figure 3-14 shows the results of executing the preceding commands.

To go down a level into the `Software` key and find subkeys beginning with m, use the following commands:

```
set-location software
get-childitem m*
```

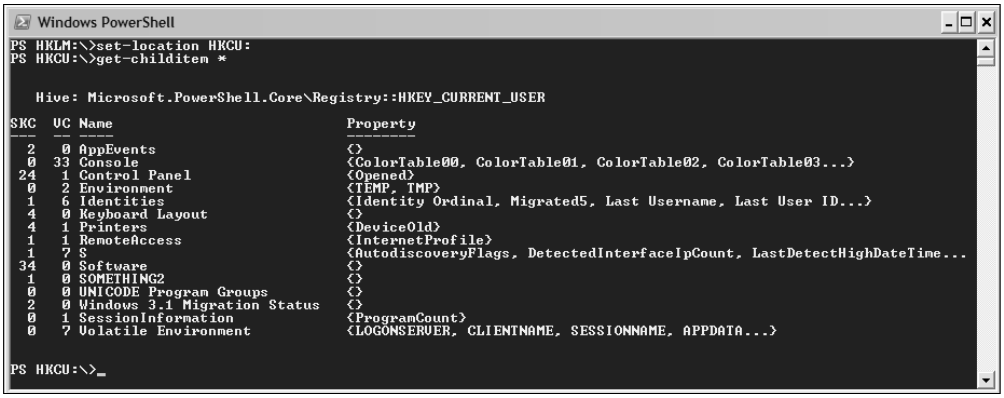


Figure 3-14

Figure 3-15 shows the results. The `set-location` cmdlet's argument is interpreted relative to the current location. Since the current working directory (using the drive metaphor) is `HKCU:\`, the `Software` key is its child, and the `set-location` cmdlet sets that as the new location. The `get-childitem` cmdlet finds the child items of that location that begin with m since `m*` uses the * wildcard to mean the character m followed by zero or more other characters.

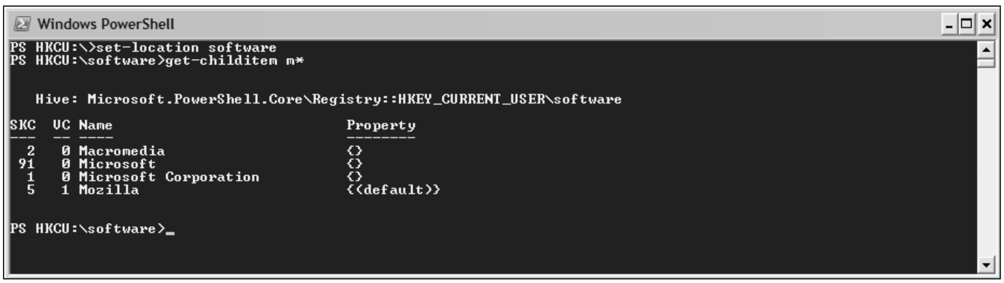


Figure 3-15

Aliases

I've already shown you some important aliases (e.g., `dir` being an alias for `get-childitem`). You can find the aliases that are available to you in your Windows PowerShell installation by using the `get-alias` command or by using the `Alias` provider. Since, the aliases are surfaced as a drive, you can simply type

```
set-location alias:
```

to navigate to the alias drive. Note that you must include the colon character in the preceding command. Then you can use the `get-childitem` cmdlet to find the available aliases. For example, to find aliases that begin with c, use this command:

```
get-childitem c*
```

Figure 3-16 shows the results.

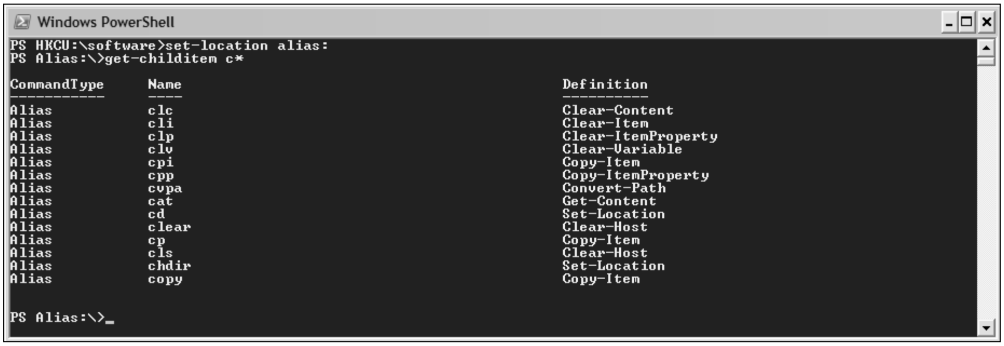


Figure 3-16

Notice in Figure 3-16 that the alias `cd` for the `set-location` cmdlet is among those displayed.

Variables

Variables, too, are surfaced as a drive. To switch to the variable drive and display variables that contain the character sequence `maximum`, use these commands:

```
set-location variable:
get-childitem *maximum*
```

The pattern `*maximum*` matches zero or more characters (as indicated by the * wildcard) followed by the literal character sequence `maximum` followed by zero or more characters. As you can see in Figure 3-17, there are several variables that contain the specified character sequence.

Figure 3-17

The `cd` alias that you saw in Figure 3-16 can be used to achieve the same result, since the `cd` alias is an alias for the `set-location` cmdlet:

```
cd variable:
get-childitem *maximum*
```

If you wanted to retrieve the information just mentioned, you could do it using a single command:

```
get-childitem variable:*maximum*
```

Active Directory

An Active Directory provider made a brief appearance in early builds of PowerShell but then was removed. It seems likely that the Active Directory provider will reappear in PowerShell some time after the release of version 1.0.

Some Exchange Server 2007 cmdlets allow you to manipulate Active Directory. At the time of writing, Exchange Server 2007 is in beta. Further information on Exchange Server 2007 and the cmdlets available in the Exchange Management Shell is available at `www.microsoft.com/exchange/default.mspx`.

Even in the absence of an Active Directory provider in Windows Powershell 1.0, you can explore and manipulate Active Directory by using the relevant .NET Framework 2.0 classes. For example, the following code finds information about Active Directory:

```
$AD = new-object System.DirectoryServices.DirectoryEntry
```

You can then use the `$AD` variable and its properties to explore Active Directory. The command

```
$AD
```

displays the root of the Active Directory hierarchy. Figure 3-18 shows the results on a test domain controller.

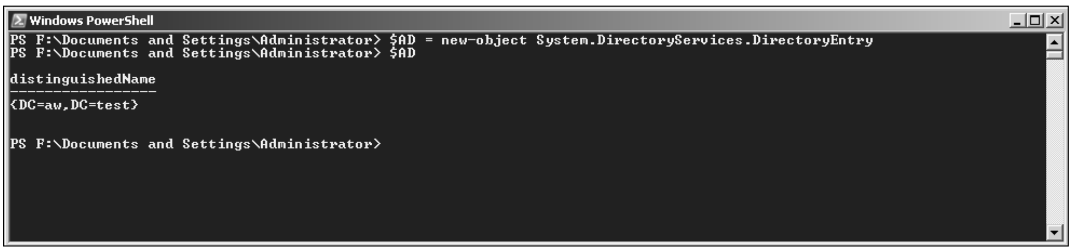


Figure 3-18

The command

```
$AD | get-member
```

displays the members of the `$AD` variable. You can then use properties such as `whenCreated` or `whenChanged` to find out when it was created or changed. For example, to find out when the Active Directory hierarchy was last changed, use this command:

```
$AD.whenChanged
```

Certificates

Certificates associate an identity with a public key and are used for purposes such as authenticating software to be installed on a network. In Windows PowerShell, you can use the `cert` drive to explore information about the certificates, if any, on a machine. For example, to move to the `cert` drive and display all child items on that drive, use this command:

```
set-location cert:
get-childitem *
```

Unlike the `variable` and `alias` drives, the `cert` drive has a hierarchy. As you can see in Figure 3-19, the Windows XP machine that the preceding command was run on has locations for the current user and local machine, with further stores contained in those.

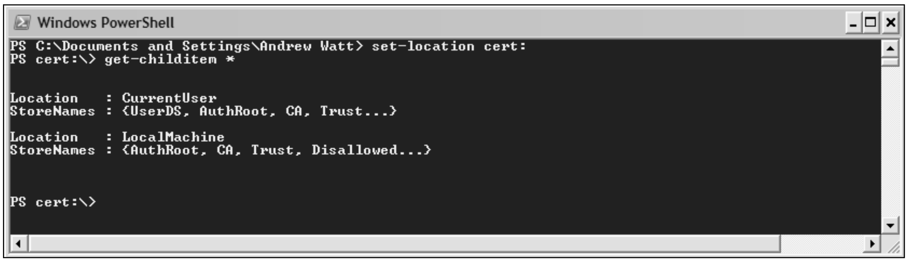


Figure 3-19

The PowerShell command shell providers generally support the same set of parameters. However, in some situations a provider can have its own parameter(s). The Certificate provider supports, for example, a `codesigning` parameter that other providers have no need for.

I discuss security in more detail in Chapter 16.

Extensibility and Backward Compatibility

Windows PowerShell uses several techniques to make things easier for users moving to PowerShell from the Windows `Cmd.exe` shell, from Unix or Linux backgrounds as well as those who use Windows GUI administration tools such as MMC, the Microsoft Management Console, version 3.

Aliases

Aliases are used in Windows PowerShell, but PowerShell is not the first command line tool to use this feature. For example, the WMI command line utility WMIC uses aliases to allow some abstraction from direct use of WMI classes. For example, to find users on a machine try this command:

```
useraccount list brief
```

The preceding command lists user accounts on a machine.

Just as WMIC aliases make it easy to access WMI functionality in a succinct way, PowerShell also supports succinct access or familiar access to a range of PowerShell functionality. Some commands, such as `cls` which is the alias for the `clear-host` cmdlet, are shorter than the underlying commands and are also familiar to many users who have used it in the `CMD.exe` shell.

By using aliases, you can simplify your use of PowerShell. For example, to retrieve information about child items of the current location, you can use the following command.

```
get-childitem *
```

But when time is pressing or you are writing multistep pipelines on the command line, it is quicker to type the following:

```
gci *
```

Or, if you are used to `CMD.exe`:

```
dir *
```

Or, if Unix commands are familiar:

```
ls *
```

As you can see from the preceding commands, there are multiple aliases for some cmdlets. Some, such as `gci`, are abbreviations of a cmdlet name, some are commands familiar from `CMD.exe`, and some are commands familiar from the Unix family of operating systems. Which you use, is up to you. If you want to create new aliases, you can use the `new-alias` cmdlet to do so. To find all aliases on the system, you can use the `get-alias` cmdlet, as in the following command:

```
get-alias
```

To display available aliases beginning with c in alphabetical order by the name of the alias, use this command:

```
get-alias c* |
sort-object Name
```

The `sort-object` cmdlet sorts objects passed to it by the first step in the pipeline. The argument to the `sort-object` cmdlet is the value of the positional `property` parameter. A fuller version of the preceding command is:

```
get-alias c* |
sort-object -property Name
```

To display available aliases sorted in alphabetical order by the name of cmdlets, use the following command. The name of the cmdlet for which an alias has been created is stored in the `Definition` property of the `System.Management.Automation.AliasInfo` objects created by executing the `get-alias` cmdlet.

```
get-alias  c* |
sort-object Definition
```

Figure 3-20 shows the results of sorting by the `Name` and `Definition` properties.

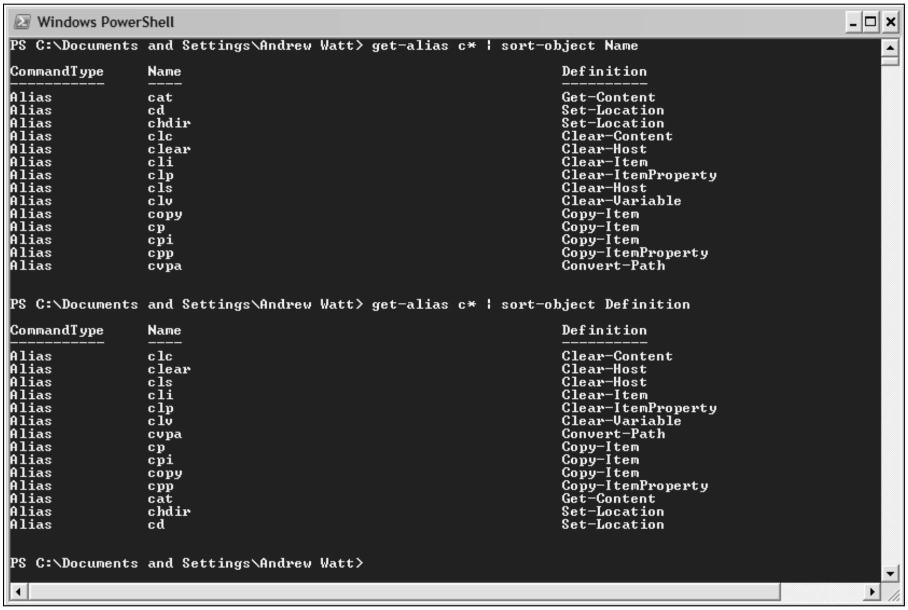


Figure 3-20

Use Existing Utilities

Windows PowerShell allows you to use familiar existing Windows command line utilities. For example, you can use the `findstr` utility to find text that matches a pattern. To demonstrate this, let's create a test document named `PowerShell.txt` with three lines of text, as follows:

```
Windows PowerShell is a great new shell and scripting language.
Windows PowerShell used to be called Monad.
This line won't be retrieved by findstr.
```

You can find the lines that contain the word PowerShell by using the following `findstr` command from the PowerShell console:

```
findstr "PowerShell" PowerShell.txt
```

As you can see in Figure 3-21, the desired lines are displayed. Make sure that you run the command in the folder that you saved `PowerShell.txt` in.

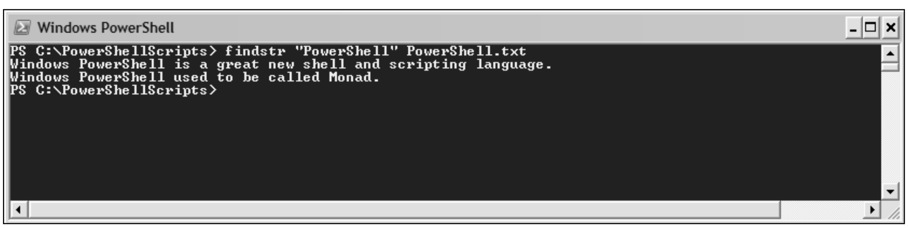


Figure 3-21

You can also use the PowerShell `get-content` cmdlet to do the same thing using this command:

```
get-content PowerShell.txt |
where-object {$_.ToString() -match ".*PowerShell.*"}
```

or, more simply:

```
get-content PowerShell.txt |
where-object {$_.ToString() -match "PowerShell"}
```

The `-match` operator allows you to use .NET regular expressions to match strings. In the preceding example, you can use either of the regular expression patterns shown. The pattern `.*` (a period followed by an asterisk) matches zero or more alphanumeric characters.

The joy of Powershell is that you can use either the PowerShell cmdlets and the like, or you can just continue to use the `findstr` utility from the PowerShell command prompt. Thus, you can continue to use familiar utilities from PowerShell command line.

You can also use the `get-command` cmdlet to discover applications on the `PATH` environment variable. For example, if you occasionally use the `findstr` utility but can't remember its exact name, you can use `get-command` to look for it using a wildcard, as in the following command:

```
get-command fi*
```

or, using the `gcm` alias:

```
gcm fi*
```

By default, `get-command` returns all types of "command" including cmdlets and executable applications. If you know that you are looking specifically for an application (rather than, say, a cmdlet) add the `CommandType` parameter as follows:

```
get-command fi* -CommandType Application
```

Figure 3-22 shows the results of executing the two preceding commands. In this case, the same commands are returned by both forms of the command, since no cmdlets begin with `fi`.

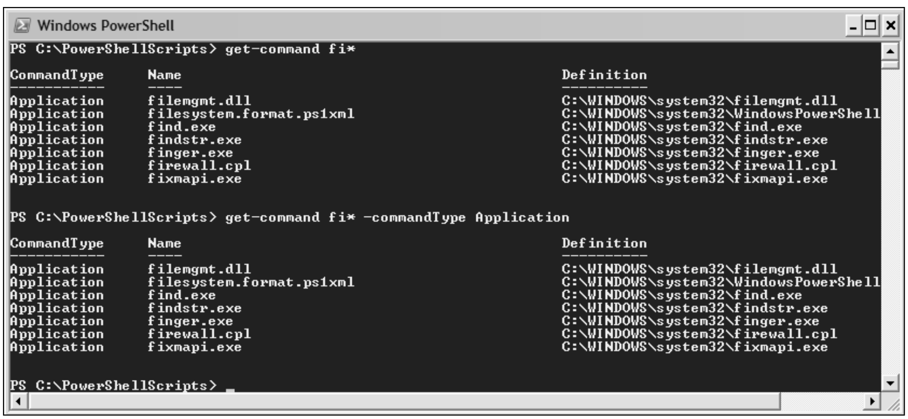


Figure 3-22

Use Familiar Commands

PowerShell makes it easy for users of `CMD.exe` to migrate over. The `get-childitem` cmdlet is the native PowerShell cmdlet used to obtain the child items of a given object. Unix admins, searching for files in a folder, might like to use the `ls` aslias, while a Windows admin might type `dir`. PowerShell supports both commands as built-in aliases. Behind the scenes, PowerShell uses the aliasing process to map the `ls` and `dir` commands to the `get-childitem` cmdlet.

Figure 3-23 shows part of an example profile file which you can use to set aliases at PowerShell startup. I have highlighted the line which sets `ls` as an alias for the `get-childitem` cmdlet. The line that sets the `dir` alias for `get-childitem` is further down in the figure.

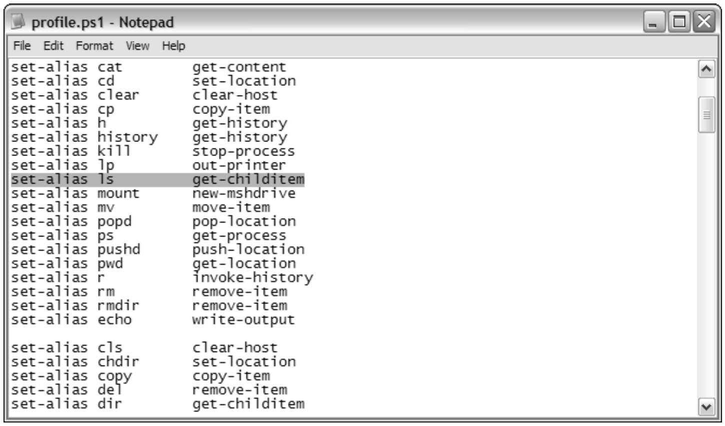


Figure 3-23

Notice that the `set-alias` cmdlet is used to create the aliases shown in Figure 3-23. Aliases and startup files are discussed in greater detail in Chapter 5.

Long Term Roadmap: Complete Coverage in 3 to 5 Years

At the time of writing, the first version of Windows PowerShell, version 1.0, has been released. Like most ambitious projects, Windows PowerShell is going to take several more years to achieve complete coverage of all the desired functionality. PowerShell version 1.0 covers many of the common things that you would want to do in administering a Windows system. Over the next 3 to 5 years, there will be further versions of PowerShell that will add further functionality, including a better development and shell environment, better remoting, and so on.

Since the coverage achieved in PowerShell version 1.0 is only part of what will come later, to carry out necessary admin tasks, you may need to fill in the gaps using COM objects, Windows Management Instrumentation (WMI), or direct manipulation of .NET classes or objects.

COM Access

Windows PowerShell provides you with the ability to access COM objects by using the `new-object` cmdlet used with the `ComObject` parameter. Once created, use COM automation to make use of the object. Microsoft expects that, over time, you will use this functionality less and less as PowerShell or third-party developers add additional cmdlets in succeeding versions.

WMI Access

Windows PowerShell also provides full read access to Windows Management Instrumentation (WMI). An important area where WMI access fills a gap in the current Powershell cmdlets is access to remote machines. The core version of PowerShell 1.0, for example, provides no cmdlets to access remote machines, except by using WMI. When you use WMI, you can achieve remote access using the `get-wmiobject` cmdlet.

Some of the cmdlets being built into Exchange Server 2007, on the other hand, can access remote machines, since Exchange Server 2007 cmdlets are designed to do this.

.NET Class Access

PowerShell version 1.0 provides very succinct ways to manipulate a subset of .NET objects. For example, writing

```
get-date
```

is more succinct than

```
[System.DateTime]::Now
```

as a way of retrieving the current date and time. For areas of functionality where no PowerShell version 1.0 cmdlet exists, you have the option of directly using the members of .NET classes or objects with the syntax I showed you earlier in this chapter. But if a cmdlet existed, it would likely be an easier or more succinct approach. In time, additional cmdlets may provide more functionality. But the PowerShell syntax that supports using .NET objects means that you are not stuck waiting for future cmdlets to be developed. You can create your own cmdlets or you can directly manipulate .NET Framework 2.0 objects.

Object-Based Approach in PowerShell

One important feature in PowerShell is that it is object-based. PowerShell operates on .NET, COM, and WMI objects. Everything in PowerShell is object-based.

Object-Based Pipelines

Command shells such as `CMD.exe` on Windows or BASH/CSHELL in Linux/Unix makes use of pipelines. A pipeline allows the result of one command to be passed to another command. `CMD.exe`, in common with the usage of Linux/Unix pipelines, typically passes strings from one command to the next. This is very useful, but it imposes the burden of string parsing on the user. In the Linux environment, the need to achieve increasingly complex string manipulation led to the development of utilities such as `awk`, `sed`, and `grep`, as well as Perl. You have immense flexibility when using that approach but at the expense of needing to learn multiple complex tools and languages, each with overlapping functionality.

The approach taken in Windows PowerShell is different, and better, in that .NET objects, not strings, are passed between steps in the pipeline. In the case of PowerShell, each command is typically a cmdlet, although you can also use .NET classes and their methods and properties in pipeline steps. Each object passed along a pipeline has the methods and properties common to that type of object. This enables you to use object notation to retrieve or manipulate desired components or values of each object.

One advantage of the object-based approach is that the displaying of the information contained in objects can also be handled as a pipeline step. The `format-table` and `format-list` cmdlets are among the ways to display information exiting a PowerShell pipeline.

A Consistent Verb-Noun Naming Scheme

Windows PowerShell cmdlets consistently use a verb-noun (that is a verb followed by a hyphen followed by a noun) naming scheme. For example, to find the running services on a machine, you can type:

```
get-service
```

This returns information about all services on the machine (whether they are running or stopped).

As a more complex example, to find the verbs available in the version of PowerShell on your machine, use this command:

```
get-command -CommandType cmdlet |
group-object verb |
sort-object Count |
format-list Count, Name,Group
```

Figure 3-24 shows part of the results on the machine I am using to write this book. As you can see, the `get` verb is frequently used, as are `new` and `set`.

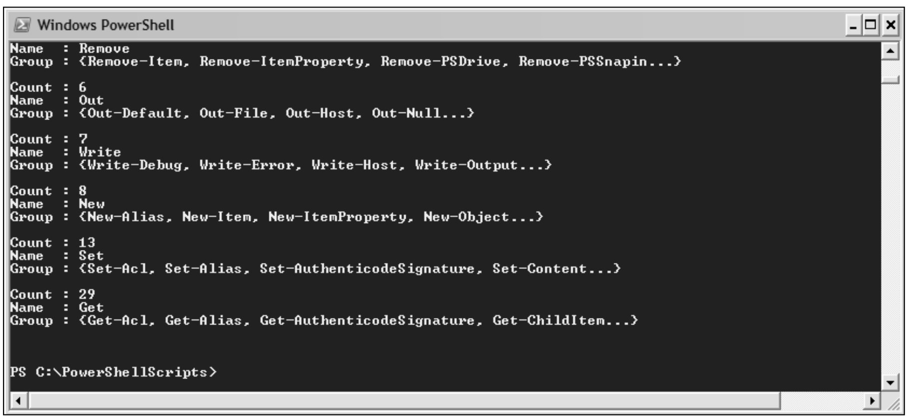


Figure 3-24

The first step of this example retrieves objects representing all available cmdlets. The second step uses the `group-object` cmdlet to group the cmdlets by the verb part of the cmdlet name. The third step sorts the groups by the count of the group (in this case the verb). The final step uses the `format-list` cmdlet to display the count of each group, its name, and the cmdlets in each group.

Coping with a Diverse World

One of the difficulties facing any administrator is that the software world is immensely varied and is constantly changing.

In this book, as is true with most computer books, all the code is tested on one or more systems. In the real world, not every machine is set up identically. So in a book on, say, Windows Server 2003, I might tell you what behavior to expect but that statement is good only at the time it is written and for the setup or setups that I test. Why? Sometimes books are written against betas of a product, and the product team make late tweaks intended to improve the product or remove a bug. Either intentionally or unintentionally, the behavior of the system may be subtly or overtly changed.

Similarly, Microsoft is putting out updates for Windows that you can apply automatically using Windows Update or at a time of your own choosing. Many of those minor updates are intended to fix security problems, but some will affect aspects of your Windows system so that they behave differently from when the book was written. Sometimes that will affect you significantly, sometimes not.

Windows PowerShell helps to reduce the kinds of uncertainty that I mentioned in the preceding paragraphs and allows you to explore the system's actual state at the time you test it. You don't need to worry that your system might differ in some characteristic from the one I or any other author tested code on. You can find out exactly the state of your machine using PowerShell.

Upgrade Path to C#

The syntax of PowerShell has been designed with a view to providing a fairly easy upgrade path to C# code. The delimiters for script blocks, for example, are paired curly braces, corresponding to the use of paired braces as delimiters in C#.

Scripting PowerShell is discussed in more detail in Chapter 10.

Working with Errors

Let's suppose that you are running a PowerShell script on a hundred or a thousand machines. On many of the machines, it's likely that the script will run without error. On a subset of the machines you will, unless you are very lucky, get some kind of error. It's just how the real world is — not everything works as you hope it will. The architects of Windows PowerShell recognized that reality, so the PowerShell approach, which is to allow you to work as if errors are expected, reflects the kind of situations that will arise in any large multiuser environment.

PowerShell allows you to use error information in several ways. I discuss errors and how you can handle them in Chapter 17.

Debugging in PowerShell

When you're writing code of any length, it is important to be able to debug the code you have written. Since PowerShell 1.0 script code isn't, at least natively, written in a GUI environment, the PowerShell shell needs to support debugging in a command line environment.

I discuss debugging in Chapter 18.

Additional PowerShell Features

In this section, I introduce some additional features of PowerShell that can affect how you use it in several situations.

Extended Wildcards

PowerShell includes support for extended wildcards in the values for cmdlet parameters, although not all parameters allow wildcards. The following table shows the wildcards supported and briefly explains their meaning.

Wildcard	Description
`?`	Matches exactly one character in a specified position.
`*`	Matches zero or more characters.
`[abc]`	Matches a class of characters. One match is found for any of the characters inside the square brackets.
`[a-c]`	Matches a range of characters. One match is found for any character between the character before the hyphen and the character after the hyphen.

In addition to supporting an extended range of wildcards, Windows PowerShell also supports regular expressions.

Wildcards are useful when, for example, you want to see what files of a particular type are present in a directory. You can of course do that using Windows Explorer, by right-clicking in a folder, selecting Arrange Icons By ➪ Type. You still have to scan a potentially large number of files to find the files of the desired type.

In PowerShell you can access all the files in the folder `C:\Windows\System32` by typing the command:

```
get-childitem -Path C:\Windows\System32
```

This returns a large number of files. To focus only on DLLs whose name begins with the character sequence `ad` (that is an `a` followed by a `d`), use this command:

```
get-childitem -Path C:\Windows\System32\ad*.dll
```

Figure 3-25 shows the result of executing the preceding command on a Windows XP machine.

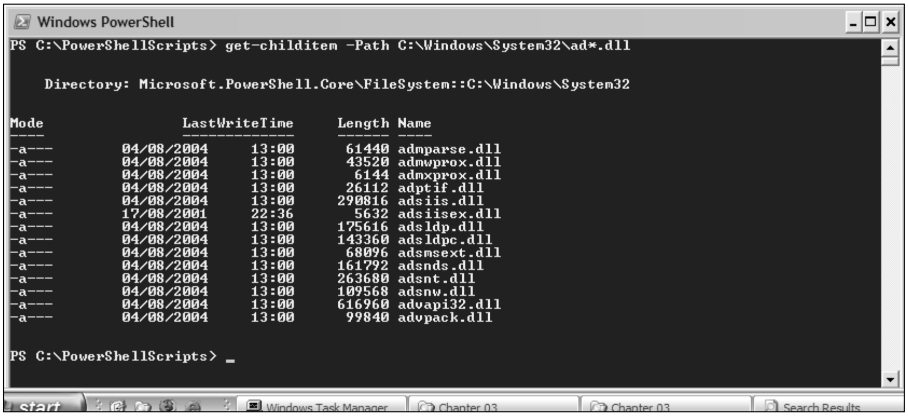


Figure 3-25

Not all cmdlet parameters support wildcards. The help files for individual cmdlets allow you to find out if a parameter does or does not support wildcards. To display the full detail of help information on parameters you need to use the `-full` parameter. For example, to view the help information about the `get-member` cmdlet, type:

```
get-help get-member -full
```

Figure 3-26 shows the help information for the `Name` and `InputObject` parameters of the `get-member` cmdlet. Notice that the `Name` parameter accepts wildcards, and the `InputObject` parameter does not.

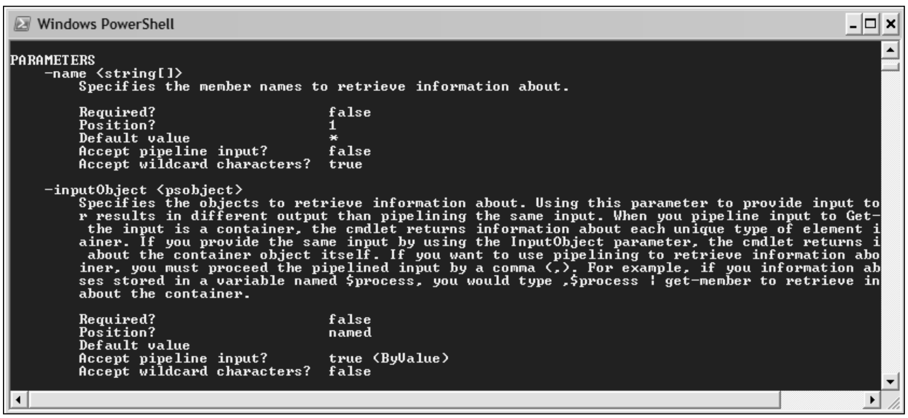


Figure 3-26

Automatic Variables

When you run the Windows PowerShell, a number of variables are set by the command shell. I list these variables in the following table.

Variable	Description
`$$`	Contains the last token received from the last line of code received by the command shell.
`$?`	Contains the success/fail status of the last operation carried out by the command shell. Holds the boolean value `True` or `False`.
`$^`	Contains the first token received from the last line of code received by the command shell.
`$_`	Contains the current pipeline object. Used by the `where-object` cmdlet, for example.
`$Args`	An array of the parameters, not explicitly defined by name, passed to a function.
`$ConfirmPreference`	Specifies what to do before PowerShell carried out an action that has side effects.
`$ConsoleFileName`	The name of the current console file.
`$DebugPreference`	Specifies the debugging policy.
`$Error`	An array of error objects.
`$ErrorActionPreference`	Specifies how errors are to be responded to.
`$ErrorView`	Specifies the mode for displaying errors.
`$ExecutionContext`	Specifies the execution objects available to cmdlets.
`$False`	The boolean value `False`.
`$FormatEnumerationLimit`	Specifies the limit on the enumeration of `IEnumerable` objects.
`$Home`	Specifies the home directory for the current user.
`$Host`	Contains information about the PowerShell console.
`$Input`	Specifies the input to a script block in a pipeline.
`$MaximumAliasCount`	Specifies the maximum number of aliases allowed.
`$MaximumDriveCount`	Specifies the maximum number of drives allowed.
`$MaximumErrorCount`	Specifies the maximum number of errors stored in the `$Error` array.
`$MaximumFunctionCount`	Specifies the maximum number of functions allowed in a session.
`$MaximumHistoryCount`	Specifies the maximum number of PowerShell commands stored in history.
`$MaximumVariableCount`	Specifies the maximum number of variables available in a session.
`$MyInvocation`	Contains information about how a script was called.
`$NestedPromptLevel`	The level of nesting of a PowerShell prompt. The level is 0 for the outermost shell.

Table continued on following page

Variable	Description
`$null`	The `NULL` value.
`$PID`	The process ID for the PowerShell `process.$pi`.
`$Profile`	The location of a user's profile file (`Profile.ps1`).
`$ProgressPreference`	Specifies the action taken when progress records are delivered.
`$PSHome`	The directory that PowerShell is installed into.
`$PWD`	The current (or present) working directory.
`$ReportErrorShow ExceptionClass`	When set to `TRUE (1)` causes the exception class for exceptions to be displayed.
`$ReportErrorShow InnerException`	When set to `TRUE (1)` causes the chain of inner exceptions to be displayed.
`$ReportErrorShowSource`	When set to `TRUE (1)` causes the assembly names of exceptions to be displayed.
`$ReportErrorShow StackTrace`	When set to `TRUE (1)` causes the stack trace for exceptions to be displayed.
`$ShellId`	Name of the PowerShell shell running (default is `Microsoft .PowerShell`.
`$True`	The boolean value `TRUE`.
`$VerbosePreference`	Specifies the action to take when the `write-verbose` cmdlet is used in a script to write data.
`$WarningPreference`	Specifies the action to take after text is written using the `write-warning` cmdlet.
`$WhatIfPreference`	Specifies whether or not `-whatif` is enabled for all commands.

Summary

Windows PowerShell version 1 provides a new command shell and scripting language for the Windows platform. PowerShell is based on the .NET Framework version 2.0. An important difference from other command shells is that PowerShell passes objects, specifically .NET objects, between steps in a pipeline.

PowerShell allows you to use your existing skills with Windows command line tools and in scripting COM objects and Windows Management Instrumentation.

Data stores, including the registry and environment variables, are exposed as drives in PowerShell. This allows you to use the same PowerShell commands to navigate those data stores as you use to navigate or manipulate the file system.

4

Using the Interactive Shell

One of the most frequent and useful ways to put Windows PowerShell to work is from the Windows PowerShell command line. For example, when you use Windows PowerShell to perform ad hoc diagnostics on a system, you will typically use it interactively from the Windows PowerShell shell command line. To diagnose causes of unusual system behavior effectively, you need to find out what the conditions are that are likely causing problems. To do so, you need to explore the characteristics of the system in an interactive way, which is where the command line comes in. The information that you discover about one aspect of the system's operation can help you focus subsequent commands that you issue. Of course, in some situations you may need to carry out similar diagnostic operations on multiple systems, and it makes sense to save at least some of the commands you use on the command line to a Windows PowerShell script file.

In this chapter, I show you how Windows PowerShell parses characters entered at the command line, so that you can understand the differences between *command mode* parsing and *expression mode* parsing. I also show you how to use Windows PowerShell commands to explore, from the command line, important information about the running of a Windows system.

Windows PowerShell's Two Command Line Parsing Approaches

One of the potentially confusing aspects for newcomers to Windows PowerShell when using the command line is that, on the command line, Windows PowerShell can parse in two ways: *command mode* and *expression mode*. To illustrate this, type the following at the command line:

```
2 + 2
```

The value 4 is displayed in the console. The expression `2 + 2` has been evaluated and then the result of the evaluation has been displayed in the console. This is *expression mode*. Next, type the following at the command line:

```
write-host 2 + 2
```

This time Windows PowerShell just wrote the string `2 + 2` to the console — Windows PowerShell performs no calculations. Figure 4-1 shows the two effects.

Figure 4-1

The preceding example demonstrates *expression mode* parsing for the first command and *command mode* parsing for the second. What do you do if you want Windows PowerShell to evaluate `2 + 2` in the second example? You can simply rewrite it as:

```
write-host (2 + 2)
```

Inside parentheses, Windows PowerShell's parser makes a fresh start for deciding which parsing mode to use. The `2 + 2` inside the paired parentheses is treated as an expression and is parsed in expression mode, just as when `2 + 2` is written alone on the command line. So, the expression inside the paired parentheses is evaluated first, and the value that results is the argument to the `write-host` cmdlet, which writes the value, `4`, to the console.

To know whether Windows PowerShell will operate in command mode or expression mode, you need to understand the rules that the parser uses to decide which mode to use.

The Windows PowerShell parser uses expression mode when the command

- Begins with a number: `2 + 2`.
- Begins with a dollar sign: `$a`.
- Begins with a quotation mark: `"This is a string"`.
- Begins with a dot followed by a number: `.5`.

The existence of expression mode is convenient in contrast to other command line environments, where you need explicitly to issue a command for the result of an expression to be displayed on screen. For example, in `CMD.exe` the preceding commands would produce an error, rather than evaluating the expression. Optionally, in Windows PowerShell, you can use the `write-host` cmdlet to display the result of an expression. For example, to display a string onscreen type the following at the command line:

```
write-host "Hello world!"
```

The Windows PowerShell parser uses command mode when the command:

- Begins with an alphabetic character: `write-host 3 + 2`.
- Begins with a dot followed by a space character: `. "myScript.ps1"`.
- Begins with a dot followed by an alphabetic character: `.someCommand`.
- Begins with an ampersand: `& something`.

Expression Mode Examples

To know whether Windows PowerShell will operate in command mode or expression mode, you need to understand the rules that the parser uses to decide which mode to use. This section covers the rules and provides examples of expression mode parsing.

The Windows PowerShell parser uses expression mode when the command

- Begins with a number

  When the initial character on a line is a number, the Windows PowerShell parser works in expression mode. So, typing

```
8 / 4
```

  at the command line causes the expression `8 / 4` to be evaluated. The result, `2`, is displayed in the console.

- Begins with a dollar sign

  When the initial character on a line is a dollar sign, then expression mode is used. For example, typing

```
$a = "Hello" + " world!"
```

  concatenates two string values, `Hello` and `world!`, and then assigns the concatenated value to the variable `$a`. You can demonstrate that the two string values were concatenated by typing

```
$a
```

  at the Windows PowerShell command line. The concatenated string is the value of the variable `$a`. The Windows PowerShell parser, in expression mode, evaluates the expression `$a` to the concatenated string and then displays that value on screen.

- Begins with a quotation mark

  Similarly, if the initial character is a quotation mark, then Windows PowerShell uses expression mode. For example, to concatenate two strings you can type

```
"Hello" + " world!"
```

  and Windows PowerShell echoes the concatenated string, `Hello world!` to the console.

- ❑ Begins with a dot followed by a number

  When the first character on a line is a dot followed by a number, Windows PowerShell uses expression mode. For example, typing

```
.75 + 1.5
```

  at the command line causes the value 2.25 to be displayed, after the two numeric values in the expression have been added together.

Figure 4-2 shows the preceding examples run in Windows PowerShell.

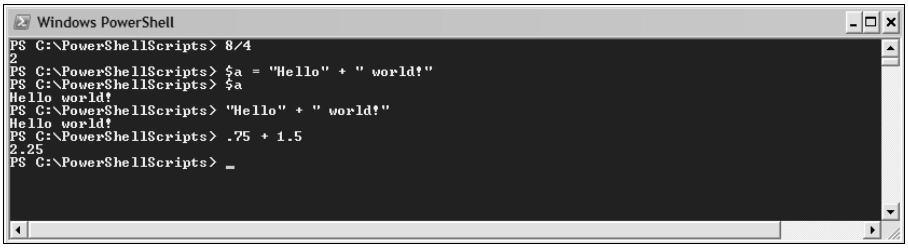


Figure 4-2

The existence of expression mode is a convenient, compared to other command line environments, where you need explicitly to issue a command for the result of an expression to be displayed onscreen. Optionally, in Windows PowerShell, you can use the `write-host` cmdlet to display the result of an expression. For example to display a string on screen, type the following at the command line:

```
write-host "Hello world!"
```

Command Mode Examples

When the first character on the line is an alphabetic character, the Windows PowerShell parser interprets everything on the line as a Windows PowerShell command. For example, if you type

```
Hello
```

Windows PowerShell produces the first error message shown in Figure 4-3. Similarly, if you mistype a Windows PowerShell command, such as using

```
get-processes
```

instead of

```
get-process
```

you will see the second error shown in Figure 4-3.

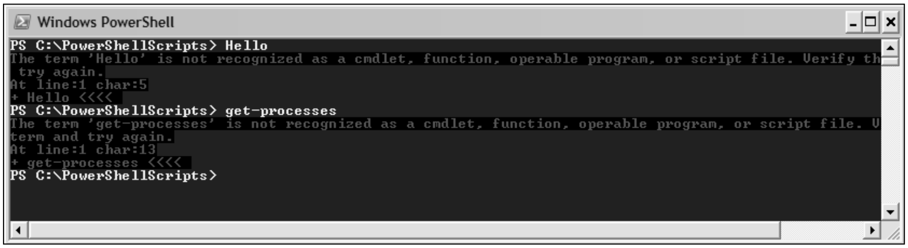


Figure 4-3

All built-in Windows PowerShell cmdlets use a singular noun in their verb-noun naming convention. So, be careful not to use a plural noun when entering the name of a cmdlet.

Of course, if you correctly type in the name of a cmdlet, for example,

```
get-process
```

the currently running processes on the machine are displayed on the PowerShell console.

Another situation in which an initial alphabetic character can be used is when you run a Windows PowerShell script. I created a very simple script named `myGetDate.ps1` and ran the script by typing

```
myGetDate.ps1
```

at the command line. This runs the script, assuming that you have saved it to a folder that is specified in the `PATH` environment variable. Internally, the script uses the Windows PowerShell `get-date` cmdlet to find the current date and time. The result is echoed to the console, as shown in Figure 4-4.

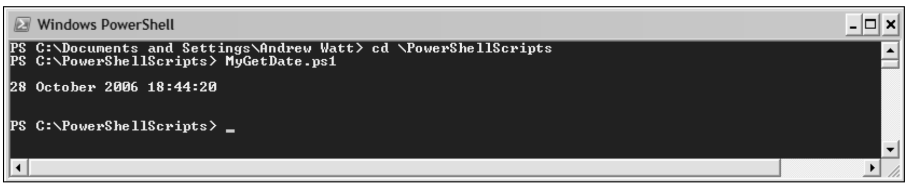


Figure 4-4

Figure 4-5

Allowing Windows PowerShell Scripts to Run

Typically, a default installation of Windows PowerShell has restrictions on the running of scripts, as a security mechanism. The available settings can prohibit all scripts, allow all scripts, or allow you to specify which digitally signed scripts can execute. To run scripts, you may need to change a registry setting. You can do that from the Windows PowerShell command line, using the `set-executionPolicy` cmdlet or using the `regedit` utility.

To change the value of the registry setting from the Windows PowerShell command line, type the following code (if you want to set the security setting for scripts to `RemoteSigned`):

```
set-executionPolicy remoteSigned
```

This will allow you to run locally created scripts without the need for signing. See Figure 4-5 to observe the effect in the registry of running the above command.

I discuss full details of security for Windows PowerShell scripts in Chapter 15. You can find help on the execution policy for running Windows PowerShell scripts by typing:

```
get-help about_signing
```

The combination of a dot followed by a space character, as in the following command,

```
. "myGetDate.ps1"
```

executes a Windows PowerShell script in the current folder whether or not the current folder is included in the value of the `PATH` environment variable. Since the purpose is to execute the script it, fits well that the Windows PowerShell parser interprets the entered characters as needing to be executed in command mode.

If a line begins with the ampersand (`&`) character then the line following the ampersand is treated as something to be executed. For example, suppose that you assign the text `get-childitem` to the variable `$a`:

```
$a = "get-childitem"
```

You can then execute the `get-childitem` cmdlet by typing:

```
&$a
```

Since the line begins with `&`, the Windows PowerShell parser interprets the line in command mode. So, the value of the variable `$a` is treated as a command, `get-childitem`, and the `get-childitem` cmdlet is executed. Figure 4-6 shows the result.

The two preceding commands (strictly the first is in expression mode, and the second in command mode) are equivalent to typing

```
get-childitem
```

at the Windows PowerShell prompt. The `get-childitem` cmdlet is similar in function to typing `dir` in the `CMD.exe` command shell.

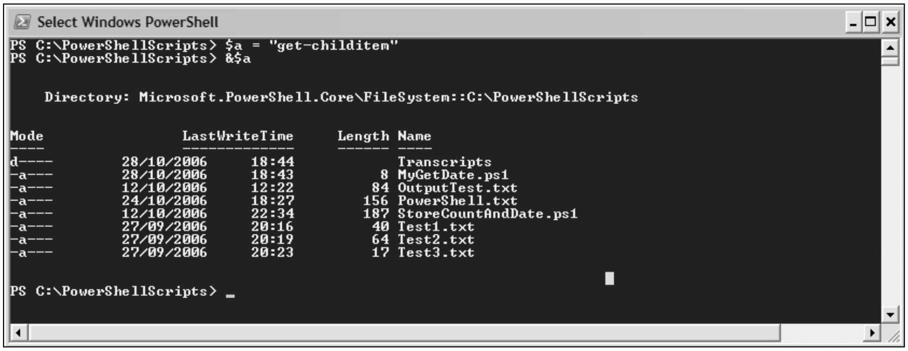


Figure 4-6

Mixing Expressions and Commands

A significant advantage of the Windows PowerShell parser having two parsing modes is that you can mix commands and expressions on the command line.

As mentioned earlier, paired parentheses create a new context for the Windows PowerShell parser to decide whether command mode or expression mode is appropriate. Parentheses can be nested to any depth. Each time a new pair of parentheses occurs, the Windows PowerShell parser reevaluates whether command mode or expression mode parsing is appropriate.

Exploring a Windows System with Windows PowerShell

In this section, I show you some techniques for exploring the current state of a Windows system using Windows PowerShell.

Finding Running Processes

The `get-process` cmdlet allows you to explore the processes running on any Windows system. For its simplest usage, just type

```
get-process
```

on the Windows PowerShell command line. This displays basic information about all currently running processes on the local machine. By default, the columns of information shown in Figure 4-7 are displayed.

> **Using Two Windows PowerShell Windows**
>
> As you begin to master Windows PowerShell, I suggest that you have two PowerShell windows open. Use one window to explore the system, and use the other to access the help system or to use the `get-member` cmdlet to list the members of Windows PowerShell objects whose use you are exploring. Also, consider ceasing to use `CMD.exe` — and use Windows PowerShell for everything you used to use `CMD.exe` for.

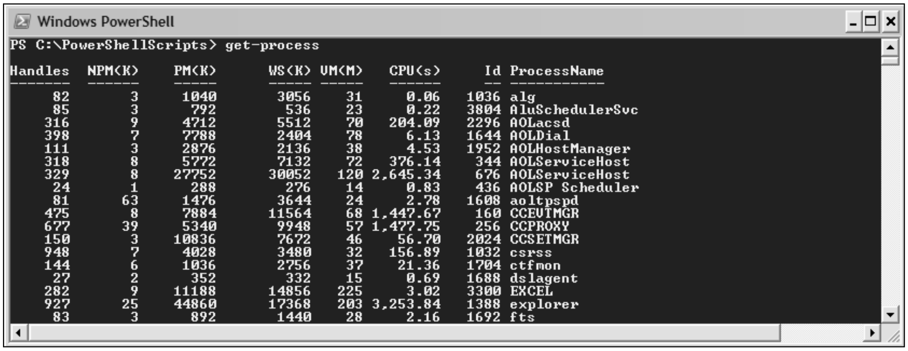


Figure 4-7

On many systems, the `get-process` cmdlet will return multiple screenfuls of information — typically on a Windows system I am running I see over 70 processes. As noted in Chapter 2, an easy way to make the output more readable is to pipe the output to More by using the following command:

```
get-process |
more
```

The results will then be displayed one screenful at a time. Press the spacebar to see another screenful of information. Press Enter to get another line of information. However, pressing Enter leads to multiple message lines being inserted between the results, such as those shown in Figure 4-8. Using the spacebar is therefore the more practical option.

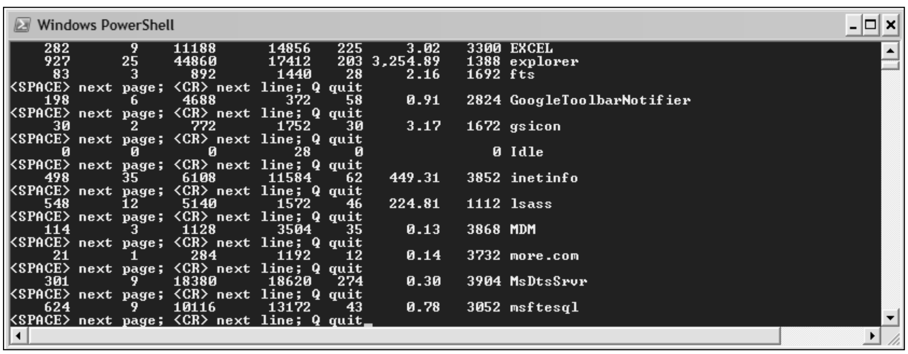


Figure 4-8

You have a number of options for filtering output. One option is to use the `where-object` cmdlet. The `where-object` cmdlet lets you filter the results returned by the `get-process` (or any other) cmdlet. The results from the `get-process` cmdlet are piped to the `where-object` cmdlet. The expression to be evaluated is contained in paired curly brackets. If an object does not satisfy the expression, it is discarded. Objects that do satisfy the expression are passed to the next part of the pipeline. In the simple example of filtering processes that follows, the matching objects are passed to the implicit default output, which is the console.

Filtering Processes Using where-object

Suppose that you wanted to see information about the instances of `svchost` that are running. You could use the following command:

```
get-process |
where-object {$_.ProcessName -eq "svchost"}
```

Be careful to include paired quotes around the name of the process that you want to filter for. If you omit the paired quotes, an error will occur since the process name is tested for equality to a string value.

Figure 4-9 shows the output on a Windows XP machine.

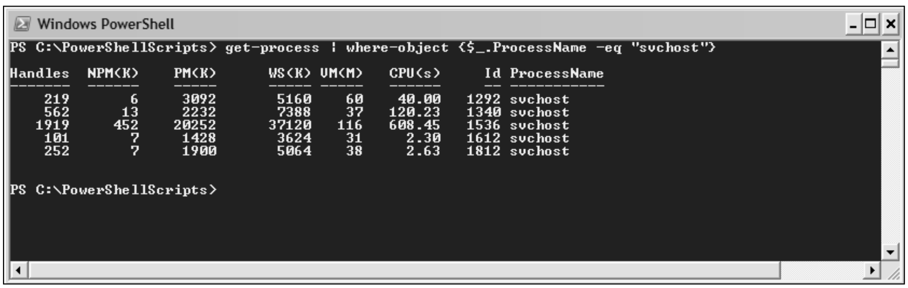


Figure 4-9

I cover the use of the `where-object` cmdlet in more detail later in this chapter.

In the preceding example, the `get-process` cmdlet retrieves information about all running processes on the system. The objects representing those processes are piped to the `where-object` cmdlet in the second step of the pipeline, where the script block contained inside the braces is used to test whether or not each object satisfies the criterion specified in the script block. Objects where the test is satisfied are passed to the next stage in the pipeline.

Let's look more closely at the contents of the curly brackets (the script block):

```
{$_.ProcessName -eq "svchost"}
```

The `$_`variable (the dollar sign followed immediately by an underscore character) represents the current object being processed in the pipeline. The specified test, whether the object's `processname` property (a string) is equal to `"svchost"`) is applied to each object returned by the `get-process` cmdlet. Since the `$`

variable is an object, use the dot notation followed by the name of a property of that object, in this case `ProcessName` to get the value of this property. The value of that property is tested for equality as indicated by the `-eq` operator. It is tested against the literal string value `"svchost"`.

> *Remember that the = operator is the assignment operator in Windows PowerShell — to test for equality in Windows PowerShell you need to use the* `-eq` *operator.*

You can use a similar technique to compare the values of other properties of a process with specified values. To find the properties of a process, use this command:

```
get-process |
get-member -membertype property |
more
```

Figure 4-10 shows the first screen of properties of the `get-process` cmdlet.

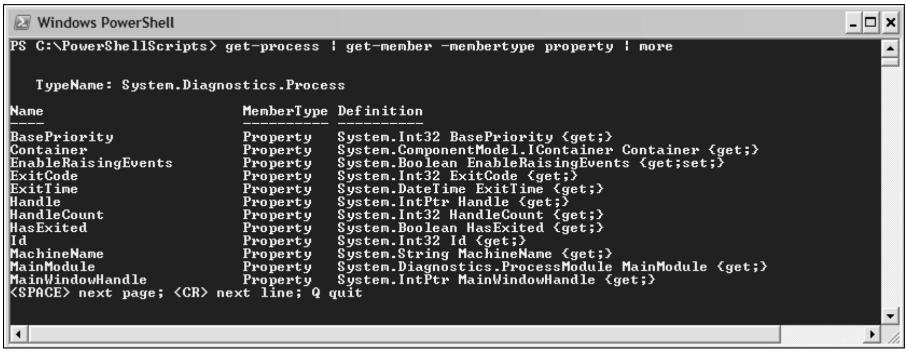


Figure 4-10

Similarly, you can find all the methods of the `get-process` cmdlet by using the following command:

```
get-process |
get-member -membertype method |
more
```

Another option for filtering output is to use wildcards. In Windows PowerShell you can use either wildcards or regular expressions, as appropriate to a particular situation. Windows PowerShell supports the following wildcards:

- `?` — Matches a single character
- `*` — Matches zero or more characters

Filtering Processes Using Wildcards

You can use wildcards when specifying a filter to be applied to objects returned from a command or pipeline step. For example, if you wanted to find information on processes that include the character sequence `sql` you could use the following command:

```
get-process "*sql*"
```

which is equivalent to:

```
get-process -processname "*sql*"
```

This command returns any process whose name consists of zero or more characters (as indicated by the first asterisk), followed by the literal character sequence `sql`, followed by zero or more characters (as indicated by the second asterisk). Stated more simply, it returns any process where the name of the process includes `sql`.

Figure 4-11 shows the results on a Windows XP machine that is running SQL Server 2005. The results include the SQL Server 2005 Full Text Search (`msftesql`), SQL Server 2005 Agent (`SQLAGENT90`), and the SQL Server 2005 database engine (`sqlservr`). On the machine in question, two instances of the SQL Server 2005 database engine are running.

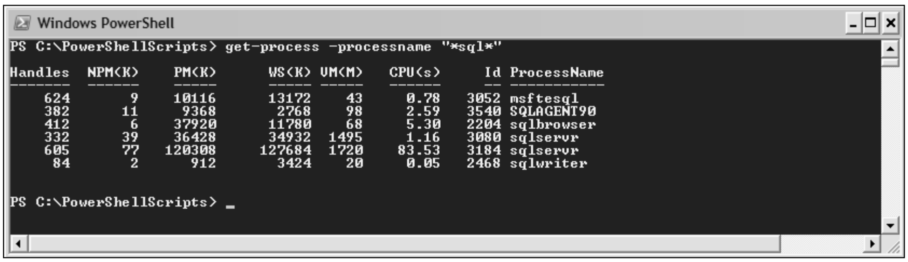


Figure 4-11

You can achieve a similar result with the `where-object` cmdlet by using the following command:

```
get-process |
where {$_.ProcessName -match "sql"}
```

Finding Out about Services

The `get-service` Windows PowerShell cmdlet allows you to explore the services available on a machine. In a simple usage (with no specified parameters):

```
get-service |
more
```

you can display all information about all the Windows services which you would normally see in the Services Microsoft Management Console snap-in, as shown in Figure 4-12. By default, the columns shown in Figure 4-12 are displayed.

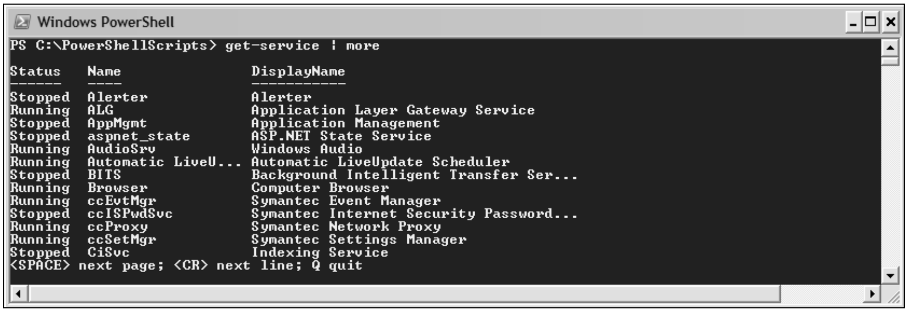


Figure 4-12

More typically, you use the `get-service` cmdlet and filter its output by using the `where-object` cmdlet.

Finding Running Services

In this example, you can find all running services on a machine by entering this Windows PowerShell command:

```
get-service | where-object {$_.status -eq "running"}
```

Figure 4-13 shows the results on a Windows XP Pro machine. Notice that the value displayed in the Status column is consistently `"running"`.

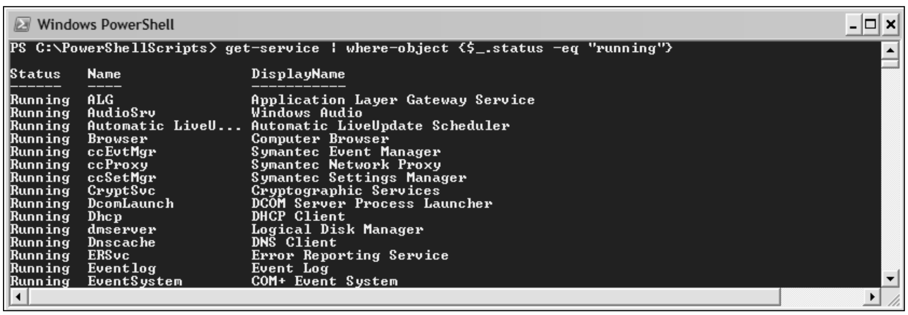


Figure 4-13

The `get-service` cmdlet returns a series of service objects — one for each service on the machine. Those objects are passed through the pipeline to the `where-object` cmdlet.

The `where-object` cmdlet filters the objects according to the conditional expression inside the paired curly brackets. The expression in this case

```
{$_.status -eq "running"}
```

specifies that the test is that the value of the `status` property in each object returned by the `get-service` cmdlet has a value equal to the string "running". In other words, only running services satisfy the test and are passed to the default formatter and so are displayed to the user.

You can, of course, look for particular running services. For example, if you want to find running services that contain the character sequence `sql` in the service's name, you could find them using either of these two commands:

```
get-service "*sql*" |
where-object {$_.status -eq "running"}
```

or:

```
get-service |
where-object {$_.Status -eq "running" -and $_.Name -match "sql"}
```

Note that the former is more efficient.

Figure 4-14 shows the results on a Windows XP Pro machine that is a developer machine for SQL Server 2005.

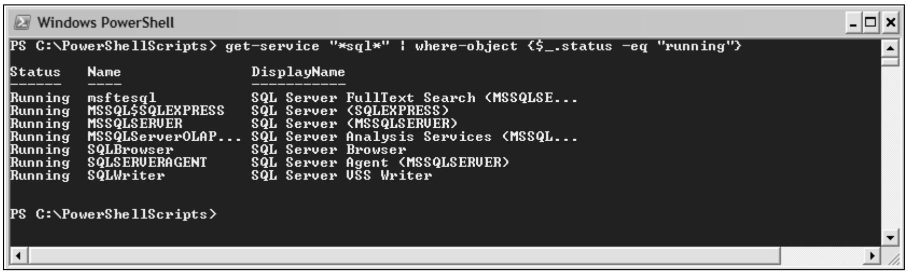


Figure 4-14

You could, equally well, find services that are stopped. To find a SQL Server–related service that is not running, you could use this command:

```
get-service "*sql*" |
where-object {$_.status -eq "stopped"}
```

By adding other conditions or filters, you can build up more complex commands if desired.

Finding Other Windows PowerShell Commands

Sometimes you may want to find Windows PowerShell commands that carry out a specific range of actions. For example, you might want to find out what commands are available that set a value. You can use the following command:

```
get-command set-*
```

to find all Windows PowerShell commands where the verb part of the cmdlet name is `set`. If you happen to have an installed application whose name begins with the character sequence `set-`, it will also be returned.

Strictly speaking, the preceding command may display commands other than Windows PowerShell cmdlets. If you need to be sure to display only Windows PowerShell cmdlets use the `-commandType` parameter in the command:

```
get-command -commandType cmdlet set-*
```

You can find all the cmdlets that contain the word “process” using this command:

```
get-command "*process*" -commandType cmdlet
```

If you are interested only in knowing about cmdlets that include “process,” it is usually safe to assume that “process” will be the noun part of a cmdlet name, so you can use the following command:

```
get-command *-process
```

Figure 4-15 shows the relevant commands using either of the two preceding approaches.

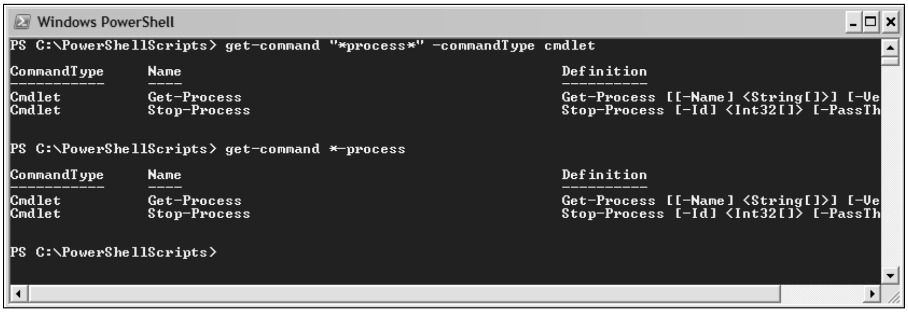


Figure 4-15

Using Abbreviated Commands

Many commands in Windows PowerShell have compact or abbreviated forms, which allow you to carry out tasks with less typing. These abbreviated commands are particularly convenient when you are working from the command line.

There are two ways to save on typing — by using command completion or by using aliases.

Command Completion

Windows PowerShell commands allow you to use the Tab key to complete commands once you have typed the verb part of a cmdlet name plus the hyphen. For example, if you want to use the `get-process` cmdlet, you can type it in fully as in the examples earlier in this chapter. However, if you type

```
get-pr
```

and then press the Tab key, you will see

```
Get-Process
```

on the screen. When carrying out tab completion, Windows PowerShell, typically, gives an initial uppercase letter to both the verb and noun parts of a cmdlet name. Since PowerShell is case-insensitive for cmdlet names this doesn't cause any undesired change in behavior.

Similarly, if you type

```
get-se
```

then press Tab, you will see

```
Get-Service
```

on the screen.

If you press the Tab key when there is more than one option available — for example when you hit Tab after typing

```
get-p
```

you will see the earliest option, as judged by alphabetical order, in this case:

```
Get-PfxCertificate
```

If you press Tab again, you will see the next option, which is

```
Get-Process
```

If you continue to press Tab, you will see the other options for command completion in that setting. Pressing Tab multiple times in this way allows you to cycle through all available cmdlets that match what you have typed.

Aliases

Aliases allow you to use short or familiar commands in place of the regular names of Windows PowerShell cmdlets. This can be useful during the period when you are learning Windows PowerShell commands, since commands that you already know may also work in PowerShell pretty much as you expect. However, aliases can also be very useful on the command line at any time when you want to save on typing.

You should be able to use the following aliases, which substitute for the `get-childitem` cmdlet.

To use the `get-childitem` cmdlet to list files and folders in a directory, simply type

```
get-childitem
```

at the command line, assuming that the current working directory is a file system directory. You will see a list of files and folders similar to the one shown in Figure 4-16. Notice that the `FileSystem` provider is used, since the current working directory is on a file system drive.

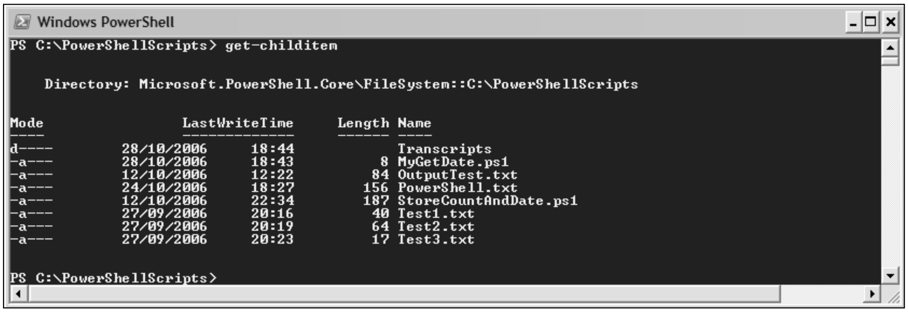


Figure 4-16

You should have two aliases available that correspond to the `get-childitem` cmdlet. First, try the conventional Windows command:

```
dir
```

to list the files and folders in the directory. You should get the result shown in Figure 4-16.

Similarly, if you are used to a Unix or Linux environment, you can type

```
ls
```

to list the files and folders in the directory. Again the results should be the same as those shown in Figure 4-16.

I discuss aliases in more detail in Chapter 5.

Working with Object Pipelines

To perform anything but the simplest tasks using Windows PowerShell, you will make use of a pipeline. A pipeline is a series of commands executed in sequence. Importantly, the objects that result from executing the first command in a pipeline are passed to the next command in the pipeline.

In this section, I will introduce several tools that you can use in pipelines.

Sequences of Commands

A pipeline is, essentially, a sequence of commands where objects from one command are passed for processing to later commands in the sequence. The separator between elements in the pipeline is the | symbol. Onscreen it is typically displayed as two separate vertical parts similar to a colon (see the command entered in Figure 4-14 for the onscreen appearance of the pipe symbol).

Filtering Using where-object

In some situations a single cmdlet may retrieve an inconveniently large number of objects. Therefore, you will often want to filter objects, for example for display or sorting. The `where-object` cmdlet allows you to filter objects according to the condition specified in a script block contained in paired curly brackets. You have seen some examples earlier in this chapter that use the `where-object` cmdlet. In addition to the `-eq` operator, which you saw used earlier in the chapter, you have many other operators available for use. These are shown in the following table.

Operator	Meaning
`-lt`	Less than
`-le`	Less than or equal to
`-gt`	Greater than
`-ge`	Greater than or equal to
`-eq`	Equal to
`-ne`	Not equal to
`-contains`	Contains
`-notcontains`	Doesn't contain
`-like`	Matches using wildcards
`-notlike`	Negated matching using wildcards
`-match`	Matches using regular expressions
`-notmatch`	Negated matching using regular expressions
`-band`	Bitwise AND
`-bor`	Bitwise OR
`-is`	Is of a specified type
`-isnot`	Is not of a specified type

By default, string comparisons are made case-insensitively. You have the option to make comparisons case-sensitive using the operators listed in the following table. In addition, there are operators that allow you specify explicitly that comparisons are case-insensitive.

Operator	Meaning
`-clt`	Case-sensitive less than
`-cle`	Case-sensitive less than or equal to
`-cgt`	Case-sensitive greater than
`-cge`	Case-sensitive greater than or equal to

Table continued on following page

Operator	Meaning
`-ceq`	Case-sensitive equals
`-cne`	Case-sensitive not equal
`-clike`	Case-sensitive matching using wildcards
`-cnotlike`	Case-sensitive failure to match using wildcards
`-ccontains`	Case-sensitive contains
`-cnotcontains`	Case-sensitive doesn't contain
`-cmatch`	Case-sensitive match using regular expressions
`-cnotmatch`	Case-sensitive failure to match using regular expressions

To find processes that contain the character sequence `sql`, you can use the following command:

```
get-process |
where-object {$_.processname -match "sql"}
```

or you can use the where alias to `where-object`, as shown here:

```
get-process |
where {$_.processname -match "sql"}
```

without changing the meaning. Either will return all processes that contain the character sequence `sql`. The `-match` operator uses regular expression matching. The character sequence `.*` (a period followed by an asterisk) matches zero or more alphabetic or numeric characters. In this particular example, you can simply the command further by omitting those character sequences:

```
get-process |
where {$_.processname -match "sql"}
```

Figure 4-17 shows the results of executing the full version of the command on a development machine running SQL Server 2005. Executing the abbreviated versions of the command returns the same results.

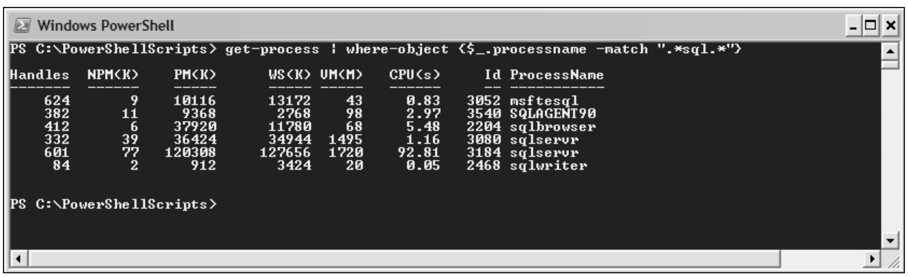


Figure 4-17

The `get-process` cmdlet returns a set of process objects representing all running processes. The results are filtered using the `where-object` cmdlet. The `-match` operator indicates that the matching process is to use regular expressions. So, the whole pattern matches zero or more characters, followed by the literal character sequence `sql`, followed by zero or more characters.

Modify the command so that you will find all services that begin with the character sequence `svc`. This will find all instances of `svchost`.

```
get-process |
where-object {$_.processname -match "^svc.*"}
```

The ^ *metacharacter* signifies a match for a position just before the first character of a sequence. In other words, it only matches when the rest of the regular expression pattern occurs at the beginning of the relevant property, in this case the `ProcessName` property. A metacharacter is a character used in a regular expression that has a meaning different from its literal significance. Then there is the literal character sequence `svc`, then the pattern `.*` (a period followed by an asterisk).

Sorting

Another task that you will frequently want to carry out is the sorting of objects in a pipeline, according to some specified criterion or criteria. The `sort-object` cmdlet allows you to sort objects produced by an earlier element in a pipeline. You can also use the `sort-object` cmdlet to sort input objects not supplied from a pipeline.

Suppose that you want to find all running processes that have a handle count in excess of 500. To find those, simply type:

```
get-process |
where-object {$_.handlecount -gt 500}
```

or:

```
get-process |
where {$_.handlecount -gt 500}
```

The default behavior is to sort the results alphabetically by the process name, as shown in Figure 4-18, which may be all you want or need.

However, you may want to find which processes are using the most handles; thus, you may find it more useful to sort the results by handle count. To do that, use the following command:

```
get-process |
where {$_.handlecount -gt 500} |
sort-object {$_.handlecount}
```

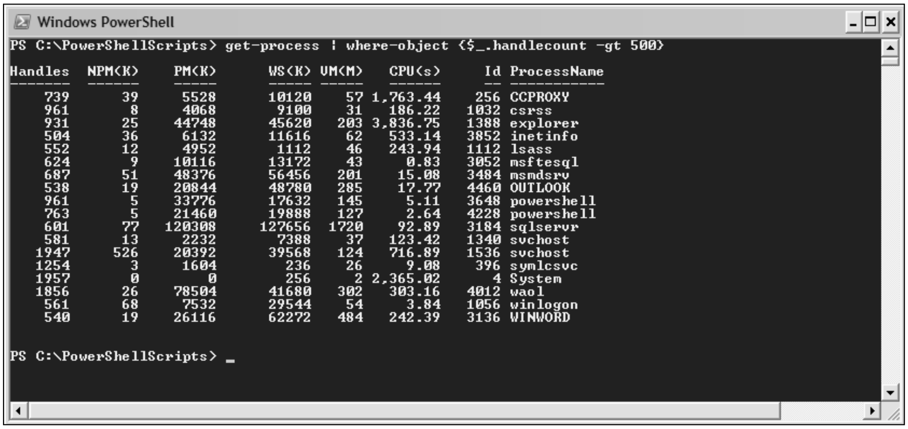


Figure 4-18

As you can see in Figure 4-19, the results are now sorted by handle count and presented in ascending order.

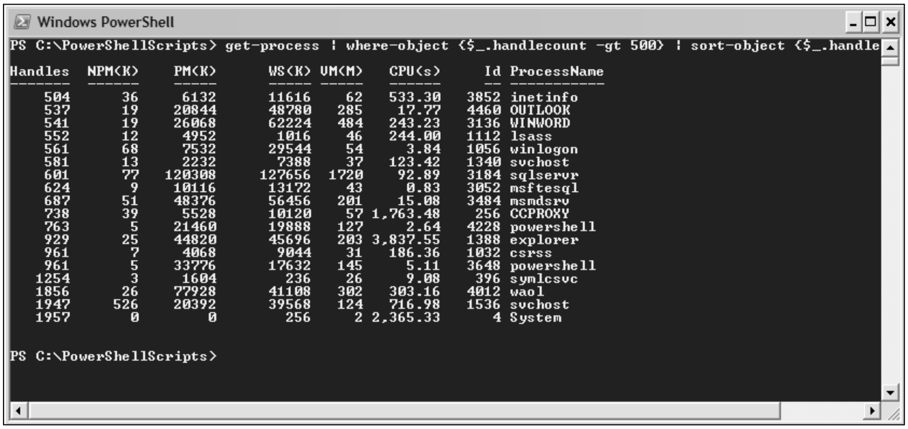


Figure 4-19

You can use an abbreviated form of the argument to the `sort-object` cmdlet in the preceding command:

```
get-process |
where {$_.handlecount -gt 500} |
sort-object handlecount
```

and the same results are displayed.

If you want to present the results in descending order (that is, with the processes with the highest handle count first), use this command:

```
get-process |
where {$_.handlecount -gt 500} |
sort-object -descending {$_.handlecount}
```

or:

```
get-process |
where {$_.handlecount -gt 500} |
sort-object -descending handlecount
```

Notice that the `-descending` parameter is used in the final element in the pipeline. You do not need to supply a value for the `-descending` parameter. In fact, if you attempt to supply a value, an error message is displayed.

Let's look at how the following code works:

```
get-process |
where {$_.handlecount -gt 500} |
sort-object {$_.handlecount}
```

The `get-process` cmdlet returns objects corresponding to all processes. The `where-object` cmdlet (the abbreviation `where` is used here) filters the processes so that only those with a handle count greater than 500 are passed to the next step in the pipeline. The `sort-object` cmdlet sorts the remaining objects according to the expression in curly brackets. In this case, it sorts the objects according to the value of the `handlecount` property of each object.

When you add the `-descending` parameter, as shown here:

```
get-process |
where {$_.handlecount -gt 500} |
sort-object -descending {$_.handlecount}
```

the objects that are returned by the first two elements in the pipeline are sorted in descending order, as specified by the expression in curly brackets, in this case the value of the `handlecount` property of each object.

Grouping

Often you will want to group results by some criterion. For example, you might want to group commands according to the verb part of the cmdlets' names. The `group-object` cmdlet allows you to group results.

In this example, I show you how to group a list of commands by the verb part of the Windows PowerShell cmdlet name.

At the command line, enter the following command:

```
get-command |
group-object {$_.verb} |
sort-object count
```

This is a three-step pipeline. The `get-command` cmdlet retrieves all commands (not only cmdlets). In the second step, objects are grouped by the value of the `verb` property. Commands other than Windows PowerShell cmdlets have no `verb` property, so the corresponding objects are discarded. The results are displayed in groups that correspond to the verb part of the cmdlet name sorted by the number of commands using this verb, as shown in Figure 4-20.

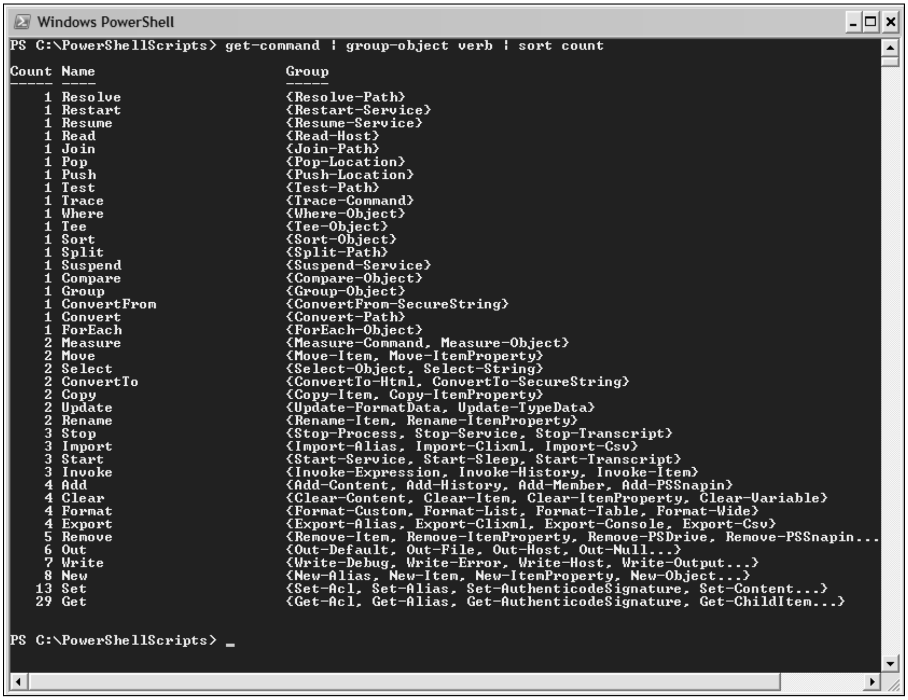


Figure 4-20

As with the `sort-object` cmdlet, the `group-object` cmdlet can accept a property name as an argument without using the paired curly braces:

```
get-command |
group-object verb |
sort-object count
```

As you can see in Figure 4-20, the default display of grouped objects can make it difficult to see what is in the group. Often you may want to follow up with other commands to look at one or more groups in additional detail.

For example, now that the previous command has shown you the verbs available in Windows PowerShell commands you might want to take a closer look at the commands that use the verb `set`.

```
get-command set-*
```

Figure 4-21 shows the cmdlets which use `set` as verb. As you can see, it is much easier to see basic information about each command in that format.

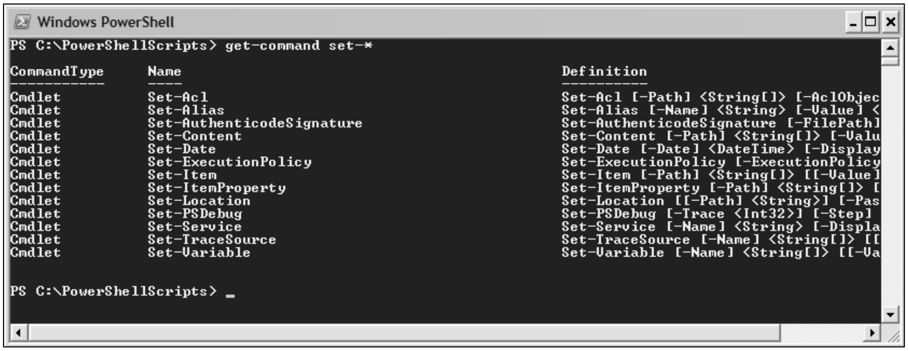


Figure 4-21

As you saw in Figure 4-20, the default formatting of grouped output is not particularly helpful with groups containing five or more objects. I discuss formatting output in more detail in Chapter 7.

Pros and Cons of Verbosity

One of the key aspects of the flexibility available to you when you construct Windows PowerShell commands is that you have options to express the same command several ways. One important reason for this is that it gives you options to create code that is quick and easy to type or that is almost self-documenting because of the "verb hyphen noun" naming convention of Windows PowerShell cmdlets.

Interactive

When you use Windows PowerShell interactively on the command line, you will often want to avoid the verbose command forms. At least, you will avoid them once you are up to speed on the verbs and nouns that make up Windows PowerShell commands.

Suppose that you want to find an alias where you already know the full command. For example, suppose that you want to find the aliases for all cmdlets whose verb is `get`. You can use the following command:

```
get-alias |
where {$_.definition -match "^get"}
```

The ^ metacharacter in the regular expression in paired quotation marks matches the position before the first character of a matching sequence of characters. More simply, the pattern `^get` matches the character sequence `get` when that character sequence occurs at the beginning of a string. The result you see will resemble Figure 4-22.

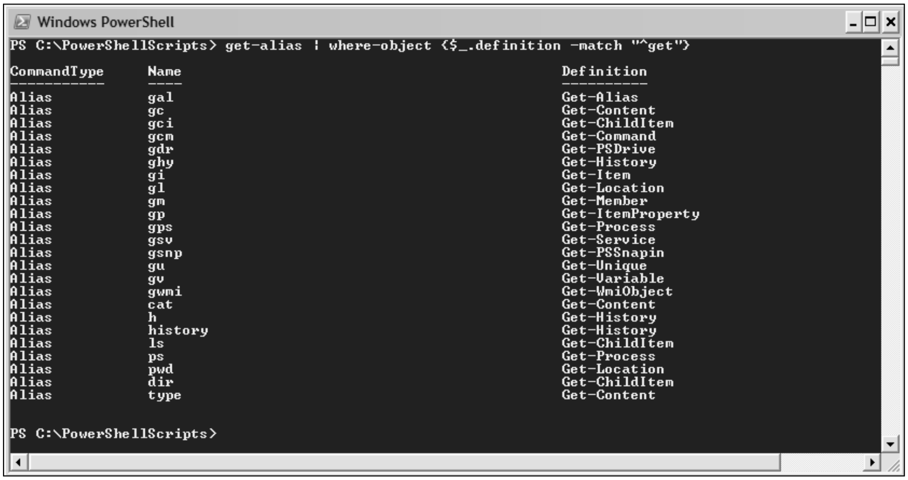


Figure 4-22

Using the `get-alias` cmdlet on its own returns all available aliases. Unfortunately, by default, it doesn't present the results in a particularly useful arrangement, since several screens of information are produced (depending on the size of your Windows PowerShell console). You can display aliases so that they are easier to read if you use this command:

```
get-alias |
more
```

You will see a listing similar to Figure 4-23.

Figure 4-23

Notice that cmdlets that use the verb `get` are separated, since the corresponding aliases are presented in alphabetical order. You can improve the usefulness of the display, as far as cmdlet verbs are concerned, by modifying the command to

```
get-alias |
sort-object {$_.definition}
```

which displays the results alphabetically by the name of the underlying cmdlet or path of an underlying application. The `definition` property of an alias object contains the name of the cmdlet.

Of course, there are circumstances that are exceptions to the general case, where you will likely want to use compact commands on the command line. This book provides one example. I want to help you grasp the full commands, so I generally provide you with the verbose form of a command, partly in order to reinforce the *verb-singularNoun* pattern of many Windows PowerShell commands. I also show pipelines with each step on a separate line, since that makes it easier for you to appreciate what each step does. In practice, when you are entering commands at the command line you will, as in the figures, likely type commands that include several pipeline steps on a single line. However, you can type a pipeline over several lines if you end each command with the | character.

Stored Commands

Using aliases and abbreviated commands is less appropriate when you write scripts. When you write Monad scripts I would strongly suggest that you use the full version of the names of Monad cmdlets. This has the advantage that your code is easier to read when it's created, since the "verb hyphen noun" naming convention is so consistent. It also has the advantage that maintenance of code is easier. Another advantage is that the code is fully portable. For example, if you use an alias in a script, you must be sure that the alias is present on all target machines. I discuss Monad scripts in greater detail in Chapter 10.

Summary

This chapter showed you the two parsing modes that Windows PowerShell uses:

- Expression mode
- Command mode

You also saw examples of using the `get-process` cmdlet to explore running processes on a Windows machine and examples of using the `get-service` cmdlet to explore services.

The chapter discussed convenience features — aliases and Tab completion — that make it easier and faster to enter commands at the Windows PowerShell command line.

Pipelines were described, as well as how you can filter (using the `where-object` cmdlet), sort (using the `sort-object` cmdlet), and group objects (using the `group-object` cmdlet) in a Windows PowerShell pipeline.

5

Using Snapins, Startup Files, and Preferences

Windows PowerShell allows you to configure several aspects that control what happens when you launch PowerShell and how PowerShell behaves after launching it. You can even add additional providers and cmdlets to those available by default.

To add further providers and cmdlets, you can load PowerShell *snapins* in addition to the core snapins that load by default when Windows PowerShell is started up. A snapin is a .NET assembly that contains Windows PowerShell providers and/or Windows PowerShell cmdlets.

You can create profile files that customize the behavior of every Windows PowerShell that you launch. Or you can customize behavior for each user individually.

You can also change the behavior of Windows PowerShell by using aliases. There are many practical advantages in PowerShell having a unique and consistent behavior. For example, once you become familiar with the verb-noun syntax convention, it becomes pretty easy to guess what the name of a command to carry out a particular task might be. However, there are also advantages in PowerShell having the flexibility to modify the behavior of the command shell to conform to user expectations or past experience. For example, by providing familiar commands (using aliases) PowerShell enables users who are familiar with other widely used command line shells to get up and running straightaway, since, at least in part, they can use commands they are already familiar with to achieve desired results.

Startup

When you execute a command such as:

```
PowerShell
```

or:

```
PowerShell -PSConsoleFile consoleFileName
```

from the command line, several things happen. The relevant console file is loaded. If no console file is specified, then the default console is loaded. If you specify a console file to load, the specified console file is loaded, if available. If not, then the default console is loaded.

A console file for Windows PowerShell version 1.0 has the suffix `.psc1`. You can create a console file to capture the current configuration settings of PowerShell, using the `export-console` cmdlet. A console file summarizes configurations for a PowerShell console. The console file is an XML file with the following basic structure:

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>1.0</PSVersion>
  <PSSnapIns />
</PSConsoleFile>
```

The behavior of the console that is loaded is open to modification by commands contained in any profile files that have been created on the machine.

Snapins

Once the default console file or a specified console file is loaded, the PowerShell snapins are loaded. A snapin is a group of PowerShell cmdlets or providers that, typically, share some functionality. You can create your own snapins or use snapins created by third parties. By default, the core PowerShell snapins are loaded. The core PowerShell snapins each have their own namespace.

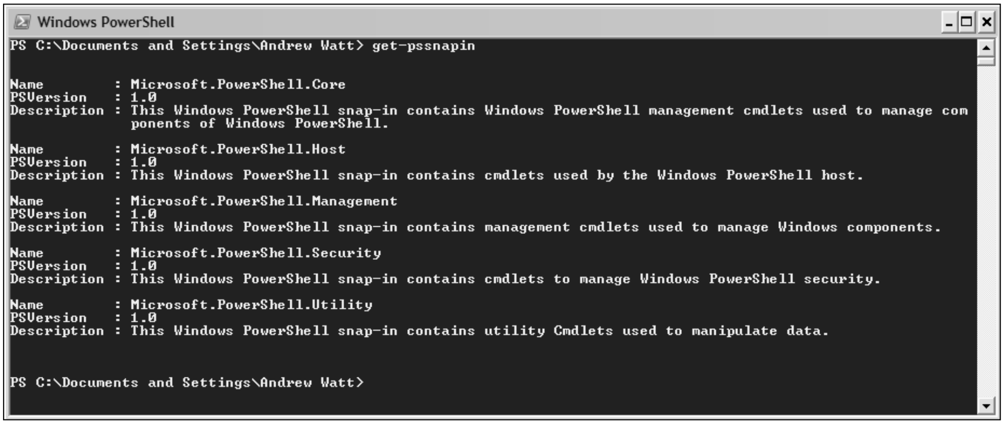
To find out which snapins are loaded in a PowerShell console, use the following command:

```
get-pssnapin
```

A list of the loaded snapins is displayed. In a typical PowerShell 1.0 installation, you can expect to see at least the following snapins:

- **Core** — Contains cmdlets that are used to affect the PowerShell engine, such as `get-help`, `get-command`, and `get-history`. Also contains the `FileSystem`, `Registry`, `Alias`, `Environment`, `Function`, and `Variable` providers. The namespace is `Microsoft.PowerShell.Core`.
- **Host** — Contains cmdlets that are used by the PowerShell host. Cmdlets include `start-transcript` and `stop-transcript`. The namespace is `Microsoft.PowerShell.Host`.
- **Management** — Contains cmdlets that are used to manage Windows components. Cmdlets include `get-service` and `get-childitem`.The namespace is `Microsoft.PowerShell.Management`.
- **Security** — Contains cmdlets that manage PowerShell security such as `get-authenticodeSignature` and `get-acl`. The namespace is `Microsoft.PowerShell.Security`.
- **Utility** — Contains utility cmdlets that manipulate data such as `get-member`, `write-host`, and `format-list`. The namespace is `Microsoft.PowerShell.Utility`.

Figure 5-1 shows the snapins on a machine with the default PowerShell installation.



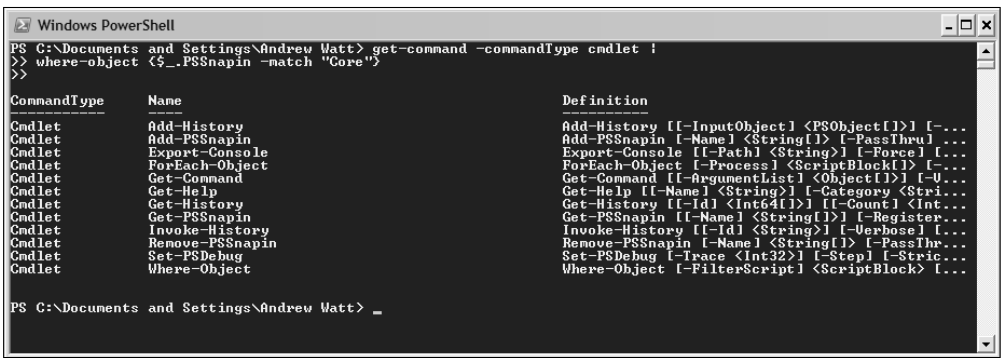
```
Windows PowerShell
PS C:\Documents and Settings\Andrew Watt> get-pssnapin


Name        : Microsoft.PowerShell.Core
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains Windows PowerShell management cmdlets used to manage com
              ponents of Windows PowerShell.

Name        : Microsoft.PowerShell.Host
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains cmdlets used by the Windows PowerShell host.

Name        : Microsoft.PowerShell.Management
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains management cmdlets used to manage Windows components.

Name        : Microsoft.PowerShell.Security
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains cmdlets to manage Windows PowerShell security.

Name        : Microsoft.PowerShell.Utility
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains utility Cmdlets used to manipulate data.


PS C:\Documents and Settings\Andrew Watt>
```

Figure 5-1

You can see which cmdlets belong to a particular snapin using a pipeline which uses the `get-command` and `where-object` cmdlets. To see the cmdlets available in the `Microsoft.PowerShell.Core` namespace, use the following command:

```
get-command -commandType cmdlet |
where-object {$_.PSSnapin -match "Core"}
```

The `get-command` cmdlet retrieves objects for all available commands. The presence of the `commandType` parameter with a value of `cmdlet` means that only objects for cmdlets are passed to the second step of the pipeline. The script block used with the `where-object` cmdlet uses regular expression matching to test whether the name of the snapin contains the character sequence `Core`. In a default install, only the `Microsoft.PowerShell.Core` snapin matches, so the cmdlets in that snapin are displayed on screen, as you can see in Figure 5-2.



```
Windows PowerShell
PS C:\Documents and Settings\Andrew Watt> get-command -commandType cmdlet |
>> where-object {$_.PSSnapin -match "Core"}
>>

CommandType     Name                                      Definition
-----------     ----                                      ----------
Cmdlet          Add-History                               Add-History [[-InputObject] <PSObject[]>] [-...
Cmdlet          Add-PSSnapin                              Add-PSSnapin [-Name] <String[]> [-PassThru] ...
Cmdlet          Export-Console                            Export-Console [[-Path] <String>] [-Force] [...
Cmdlet          ForEach-Object                            ForEach-Object [-Process] <ScriptBlock[]> [-...
Cmdlet          Get-Command                               Get-Command [[-ArgumentList] <Object[]>] [-V...
Cmdlet          Get-Help                                  Get-Help [[-Name] <String>] [-Category <Stri...
Cmdlet          Get-History                               Get-History [[-Id] <Int64[]>] [[-Count] <Int...
Cmdlet          Get-PSSnapin                              Get-PSSnapin [[-Name] <String[]>] [-Register...
Cmdlet          Invoke-History                            Invoke-History [[-Id] <String>] [-Verbose] [...
Cmdlet          Remove-PSSnapin                           Remove-PSSnapin [-Name] <String[]> [-PassThr...
Cmdlet          Set-PSDebug                               Set-PSDebug [-Trace <Int32>] [-Step] [-Stric...
Cmdlet          Where-Object                              Where-Object [-FilterScript] <ScriptBlock> [...

PS C:\Documents and Settings\Andrew Watt> _
```

Figure 5-2

If you want to demonstrate which snapins contain which PowerShell providers, use the following command:

```
get-psprovider |
format-table Name, PSSnapin
```

Figure 5-3 shows the results. Notice that in a default PowerShell install, all but the `Certificate` provider are contained in the `Core` snapin. The `Certificate` provider is contained in the Security snapin.

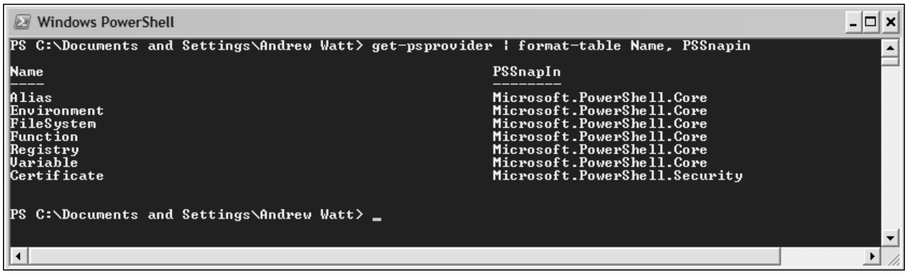


Figure 5-3

The cmdlets in the `Microsoft.PowerShell.Core` snapin are:

- `add-history` — Adds entries to a session history.
- `add-PSSnapin` — Adds one or more PowerShell snapins to the current console.
- `export-console` — Exports any configuration changes made to the current console. Using this cmdlet overwrites any existing console file of the specified name.
- `foreach-object` — Processes a set of objects according to code inside an accompanying script block.
- `get-command` — Retrieves information about a command.
- `get-help` — Retrieves help information, typically about a specified cmdlet or PowerShell language feature.
- `get-history` — Retrieves the session history.
- `get-PSSnapin` — Lists the snapins registered in a session.
- `invoke-history` — Invokes a command stored in the session history.
- `remove-PSSnapin` — Removes one or more PowerShell snapins from the current console process.
- `set-PSDebug` — Turns PowerShell script debugging on or off and, optionally, sets a trace level. (I discuss debugging in more detail in Chapter 18).
- `where-object` — Filters objects in a pipeline according to a test specified in an accompanying script block.

Remember, the names of cmdlets are treated as case-insensitive by the PowerShell parser, so you can write them all in lowercase or add uppercase characters to highlight the start of, for example, a new noun. The choice is yours.

The cmdlets in the `Microsoft.PowerShell.Host` snapin are:

- `start-transscript` — Starts a transcript of a PowerShell session
- `stop-transcript` — Stops a transcript of a PowerShell session

The cmdlets in the `Microsoft.PowerShell.Management` snapin are:

- `add-content` — Adds content to a specified item or items
- `clear-content` — Removes content from an item, often a file, but does not delete the item or file
- `clear-item` — Clears an item but does not remove it
- `clear-ItemProperty` — Removes a value from a specified property
- `convert-path` — Converts a path to a provider path
- `copy-item` — Copies an item, for example a file, to another location using a PowerShell provider
- `copy-itemProperty` — Copies a property between locations or namespaces
- `get-childitem` — Retrieves the child items of a specified location
- `get-content` — Retrieves the content of an item, for example a file, at a specified location
- `get-eventlog` — Retrieves event log data
- `get-item` — Retrieves an object which represents an item in a namespace
- `get-itemProperty` — Retrieves properties of a specified object
- `get-location` — Displays the current location
- `get-process` — Retrieves information about running processes on a machine
- `get-PSDrive` — Retrieves information about one or more drives
- `get-PSProvider` — Retrieves information about one or more PowerShell providers
- `get-service` — Retrieves information about services on a machine
- `get-wmiobject` — Produces a WMI object or lists WMI classes available on a machine
- `invoke-item` — Invokes an executable or opens a file
- `join-path` — Combines elements of a path into a path
- `move-item` — Moves an item from one location to another
- `move-itemProperty` — Moves a property from one location to another
- `new-item` — Creates a new item in a namespace
- `new-itemProperty` — Creates a new item property at a specified location
- `new-PSDrive` – Creates a new drive
- `new-service` — Creates a new service on a computer
- `pop-location` — Pops a previous location from the stack and uses it to define the current working location

- `push-location` — Pushes a location on to the stack
- `remove-item` — Deletes an item using a specified PowerShell provider
- `remove-itemProperty` — Removes a property (and its value) from a specified location
- `remove-PSDrive` — Removes a drive
- `rename-item` — Renames an existing item
- `rename-itemProperty` — Renames a property without moving it
- `resolve-path` — Resolves the wildcard character(s) in a path
- `restart-service` — Stops a service and then restarts it
- `resume-service` — Makes a suspended service resume running
- `set-content` — Sets the content in a specified item, typically a file
- `set-item` — Sets the value of an item
- `set-itemProperty` — Sets the value of a property at a specified location to a specified value
- `set-location` — Sets the value of the current working location to a specified location
- `set-service` — Sets the value of properties of a service
- `split-path` — Finds the component parts of a path and makes specified components available for pipeline processing
- `start-service` — Starts a service
- `stop-process` — Stops a process
- `stop-service` — Stops a service
- `suspend-service` — Suspends a service
- `test-path` — Tests whether or not a path exists

The cmdlets in the `Microsoft.PowerShell.Security` snapin are:

- `ConvertFrom-SecureString` — Exports a secure string to a safe, serialized format
- `ConvertTo-SecureString` — Converts a supplied normal string to a secure string
- `get-acl` — Retrieves the access control list associated with an object
- `get-authenticodeSignature` — Retrieves the signature associated with a file
- `get-credential` — Retrieves a credential object
- `get-executionPolicy` — Retrieves the PowerShell script execution policy
- `get-PfxCertificate` — Retrieves the Pfx certificate information
- `set-acl` — Sets the access control list for an object
- `set-authenticodeSignature` — Applies an authenticode signature to a file
- `set-executionPolicy` — Sets the PowerShell script execution policy

The cmdlets in the `Microsoft.PowerShell.Utility` snapin are:

- `add-member` — Adds a user-defined custom member to an object
- `clear-variable` — Removes the value from a variable without removing the variable itself
- `compare-object` — Compares two streams of objects
- `ConvertTo-html` — Converts the input to an HTML table
- `export-alias` — Exports a list of aliases to a file
- `export-clixml` — Produces an XML representation of a PowerShell object or objects
- `export-csv` — Creates a list of comma-separated values from intput objects
- `format-custom` — Formats output display in a custom way
- `format-list` — Formats output display as a list
- `format-table` — Formats output display as a table
- `format-wide` — Formats output as a customizable table
- `get-alias` — Retrieves available aliases
- `get-culture` — Retrieves information about the current culture
- `get-date` — Retrieves the current date and time
- `get-host` — Retrieves information about the current host
- `get-member` — Displays information about the members of an input object or objects
- `get-traceSource` — Displays information about trace sources and their properties
- `get-UICulture` — Retrieves information about the current UI culture
- `get-unique` — Retrieves unique items from a sorted list
- `get-variable` — Retrieves a PowerShell variable and its value
- `group-object` — Groups objects according to a specified criterion or multiple criteria
- `import-alias` — Imports a list of aliases from a file
- `import-clixml` — Imports a clixml file and builds an object from its content
- `import-csv` — Imports a comma-separated value file and creates an object or objects
- `invoke-expression` — Executes a string argument as an expression
- `measure-command` — Measures the execution time for a script block or cmdlet
- `measure-object` — Calculates measures of a property of an object such as average
- `new-alias` — Creates a new PowerShell alias
- `new-object` — Creates a new object
- `new-timespan` — Creates a timespan object
- `new-variable` — Creates a new PowerShell variable

- `out-default` — The default controller of PowerShell output
- `out-file` — Sends output to a specified file
- `out-host` — Displays pipeline output on the host
- `out-null` — Sends output to null, in effect deleting the output
- `out-printer` — Sends output to a printer
- `out-string` — Converts pipeline output to a string or strings
- `read-host` — Reads a prompted value from the host, allowing a script to capture user input
- `remove-variable` — Removes a variable
- `select-object` — Selects objects or properties based on criteria specified in its parameters
- `select-string` — Allows you to search for character patterns in a string or file
- `set-alias` — Creates a new alias
- `set-date` — Sets the system date and time
- `set-traceSource` — Configures trace sources
- `set-variable` — Sets the value(s) of a variable, creating a new variable if necessary
- `sort-object` — Sorts input objects according to a specified criterion or multiple criteria
- `start-sleep` — Suspends execution for a specified period of time
- `tee-object` — Sends objects to two destinations
- `trace-command` — Turns on tracing according to a specified configuration for a specified expression
- `update-formatData` — Updates format data files
- `update-typeData` — Updates type data files
- `write-debug` — Writes a debug message to the host
- `write-error` — Writes an error object to a pipeline
- `write-host` — Writes objects to the host
- `write-output` — Writes an object or objects to a pipeline
- `write-progress` — Writes a record of progress to the host
- `write-verbose` — Writes a string to the host's verbose display
- `write-warning` — Writes a warning message to the host

I will describe the behavior of the cmdlets in the preceding list in detail in later chapters, as well as list their parameters in full and demonstrate how they can be put to use.

Additional snapins can be loaded at startup using a console file. If no console file is specified at startup, then the cmdlets in the snapins listed earlier in this section are loaded.

Profiles

Once the console and the core PowerShell snapins have been loaded, the profile files are processed. A profile is a PowerShell script that runs automatically when Windows PowerShell starts up. It can contain commands to add aliases, define functions, and configure the console in other ways. I show you a sample profile file later in this section.

The options for profile file locations are listed here. The profile files, if present, are run in the following order:

- `%windir%\system32\WindowsPowerShell\v1.0\profile.ps1` — Sets the profile for any PowerShell console for all users
- `%windir%\system32\WindowsPowerShell\v1.0\ Microsoft.PowerShell_profile.ps1` — Sets a profile for all users but only if they are loading the default Windows PowerShell console
- `%UserProfile%\My Documents\WindowsPowerShell\profile.ps1` — Sets a user-specific profile for any PowerShell console that a specific user is loading
- `%UserProfile%\My Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1` — Sets a user-specific profile for a PowerShell console but only for the default Windows PowerShell console

If there is more than one profile file on a specific machine and there is any conflict between the commands in those files, the commands in the more specific profile file take precedence.

An administrator can set up profiles that are run for all users. To do that for the default Windows PowerShell console the profile file, `Microsoft.PowerShell_profile.ps1`, is created in the folder `%windir%\System32\WindowsPowerShell\v1.0`. Alias definitions and function definitions found in that file, if present, are used for all users of that machine.

Profile files are PowerShell script files containing statements that allow you, or an administrator, to set aliases, to declare functions, and to manipulate variables. The PowerShell executable looks in the previously specified locations for `profile.ps1` files (for a user-selected console) or `Microsoft.PowerShell_Profile.ps1` files (for the Windows PowerShell console-specific profiles). If the files are present those script files are executed, subject to the constraints of the execution policy. The default execution policy is `Restricted`. Before you run a profile when you launch Windows PowerShell, you can set the execution policy to `Signed` (in which case you need to digitally sign the profile file before it will run) or set the execution policy to `RemoteSigned` or `Unrestricted`.

By default, when you start PowerShell, the profile files mentioned in the previous section are executed, if they exist. However, you have an option to skip the execution of user profile files by using the `-noprofile` switch when you start the PowerShell command shell. So, to start PowerShell without executing user profiles simply type

```
PowerShell -noprofile
```

at the command prompt.

You can find out if a user profile has been created by typing the following command:

```
test-path $profile
```

If the user profile exists, True is displayed in the console.

You can view the location of the user profile by typing

```
$profile
```

at the command prompt. The path for the profile file is displayed.

You can open the user profile in Notepad by typing:

```
notepad $profile
```

Figure 5-4 show the results of executing the preceding commands. On the machine in question, a simple profile using the `start-transcript` cmdlet has been created.

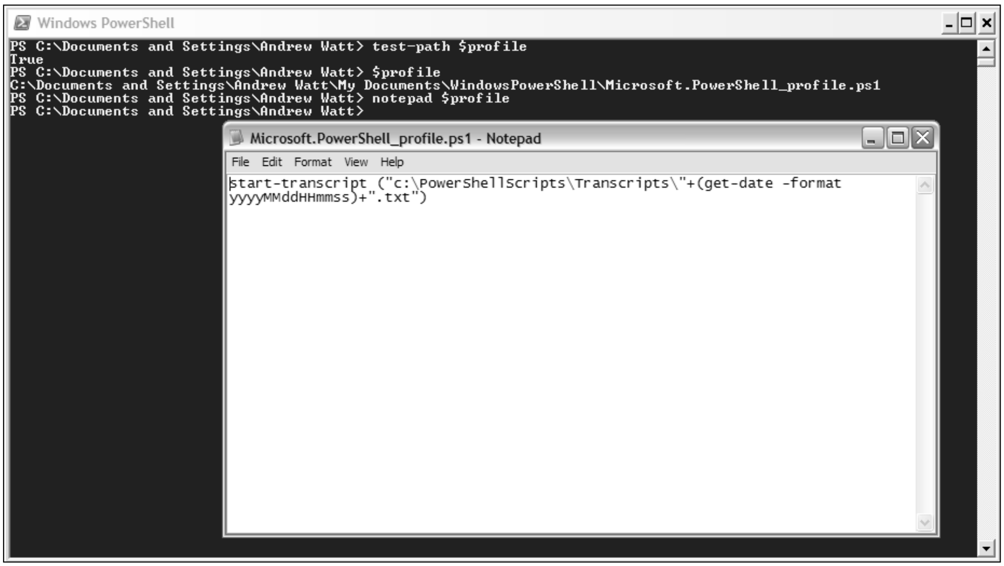


Figure 5-4

Profile.ps1

The content of a `profile.ps1` file may use the `set-alias` statement (to define aliases for a range of cmdlets) and function definitions to allow functions to replace cmdlets or provide convenient functionality. As you become more familiar with PowerShell, you are likely to add an increasing number of utility functions so that function definitions may figure increasingly prominently in your own profile files. If you create a large profile file you may find a noticeable slowing in the time to launch PowerShell.

A profile file is a PowerShell script file. If you set the execution policy to Restricted, no profile files are executed.

You use the `set-alias` cmdlet to create an alias for a cmdlet. When you use the `set-alias` cmdlet in a profile file that alias is available every time you start Windows PowerShell (subject to being overridden by a higher-priority profile file). You can use any cmdlet as the target of an alias, but not a program. You can't, for example, use an alias to substitute for the `exit` command or create an alias for the `ipconfig` program. The following command:

```
set-alias stop exit
```

causes an error message to be displayed when you attempt to use stop as an alias for `exit`.

The Example profile.ps1

Windows PowerShell comes with an example `profile.ps1` file. It is located in the `C:\WINDOWS\system32\windowspowershell\v1.0\examples` folder. The code included, apart from copyright notice and disclaimer, in the version at the time of writing is shown below. Notice that it creates a number of aliases and defines some simple functions.

```
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
    }
}
```

In this chapter, I will look in more detail at the use of the `set-alias` cmdlet.

Aliases

Windows PowerShell supports creating aliases for PowerShell commands (that is, for cmdlets), including the use of parameters. An alias is, essentially, an alternative name used to run a cmdlet.

One reason you might use aliases is that you are familiar with a particular operating system or tool and you wish to use the commands that you are already familiar with to carry out frequently performed tasks. Another reason to use aliases is to save on typing. For example, if you want to retrieve a list of all running processes, you can use the full syntax:

```
get-process
```

or with an explicit wildcard:

```
get-process *
```

Alternatively, you can use the alias in the command:

```
gps
```

or:

```
gps *
```

to achieve the same thing.

Similarly, if you want to list files contained in a folder, you can achieve that using the full PowerShell command:

```
get-childitem
```

But if you are familiar with the current Microsoft command shell, you might prefer to use:

```
dir
```

Or, if you use a Linux shell, you might prefer to use:

```
ls
```

Each of those options is shorter than the full syntax and is familiar to large numbers of administrators. These aliases are provided by default in a PowerShell install.

> **There is a potential trap in the flexibility that PowerShell gives you to create aliases if you attempt to execute scripts that contain aliases in the code but they aren't available on the machine in question. I suggest that you confine your use of aliases to your own command line work with PowerShell. Alternatively, you can agree with colleagues on an acceptable list of aliases to have available on your company's systems and include the creation of the necessary aliases in profiles on all machines. Remember the need for updating as the requirements of individuals or groups change over time.**
>
> **In general, I recommend avoiding the use of aliases in PowerShell scripts. If you are going to share scripts with other users, the use of aliases (particularly if they are nonstandard) will make the scripts more difficult to read and maintain, at least for some users. For example, not all users will be familiar with the `ls` alias, although it will be second nature to Unix administrators. If you use a custom alias, or list of custom aliases, then your scripts become either less portable (if the aliases aren't set outside the script on all machines the script will likely fail) or more cumbersome to create (you will need to add `set-alias` statements at the beginning of each script).**

In this example, I show you how to find all aliases available on your system.

To do this, remember that PowerShell exposes many features of a Windows system, including aliases, as drives, analogous to hard disk drives. To see the drives available on your system, you can type this command:

```
get-psdrive
```

at the command line. All the drives on your system are displayed, as shown in Figure 5-5.

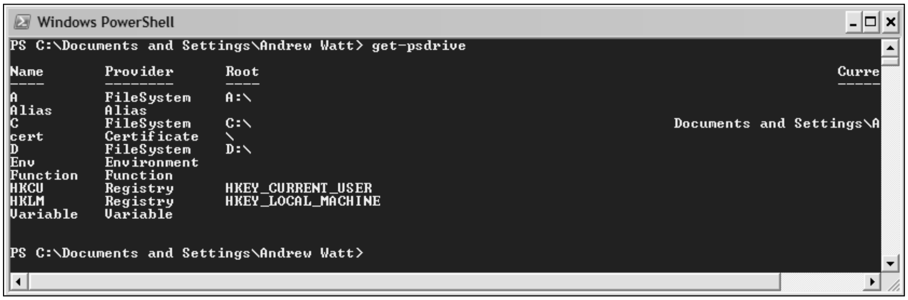


Figure 5-5

As you can see in Figure 5-5, one of the drives has the name `Alias`. You can use that drive similarly to the way you use drives and folders on a hard drive. To see all available aliases, with the results paged, type

```
cd alias:
```

to switch to the `alias` drive. Then type

```
get-childitem |
more
```

or, assuming that the `dir` alias is available on your system,

```
dir |
more
```

to display all aliases. Figure 5-6 shows one screen of results on a Windows XP machine. You may prefer to examine aliases using the `get-alias` cmdlet that I introduce later in this chapter.

The command `get-psdrive` exposes all the parts of the Windows system that PowerShell shows to the user as drives. This includes the alias drive. Under the covers, Windows PowerShell uses one of the available providers. Earlier in this chapter, I showed you how to find the available providers using the `get-psprovider` cmdlet.

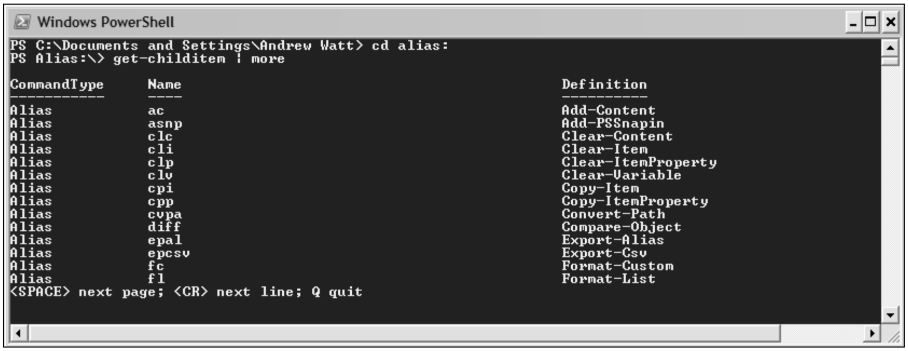


Figure 5-6

You switch to the alias drive using the command `cd alias:`. If you omit the colon at the end of the command, an error message is displayed.

The `get-childitem` cmdlet retrieves all aliases. Piping the results to `more` gives you the convenience of reading one screen of results at a time.

The default formatter displays the results in three columns: `CommandType`, `Name`, and `Definition`. The content in the `CommandType` column is the same for all aliases, so you may prefer to display only the `Name` and `Definition` columns. To do that, use the following command (assuming that you have already selected the `alias` drive):

```
get-childitem |
select-object Name, Definition |
more
```

The `select-object` cmdlet allows you to display only the desired properties of the objects of interest, in this case aliases. The aliases are not sorted, although they appear to be in the first screen of results. Inspect further screens of results to confirm that they are not ordered.

To order the aliases by name, use this command:

```
get-childitem |
select-object Name, Definition |
sort-object Name |
more
```

Figure 5-7 shows one screen of the sorted data.

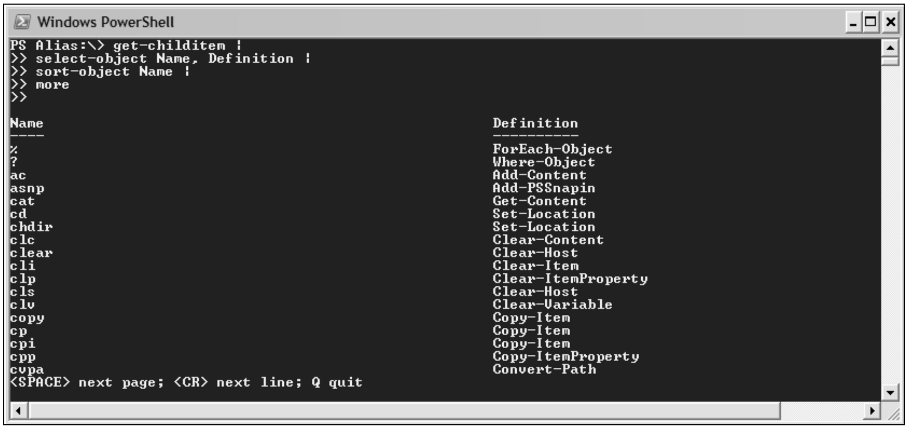


Figure 5-7

As before, the `get-childitem` cmdlet retrieves all child items in the `alias` drive from the alias provider. That is, it retrieves objects corresponding to all defined aliases.

The `select-object` cmdlet specifies which properties of the child items are to be passed along the pipeline. In this case, the `Name` and `Definition` properties are selected and passed on. I used a positional `property` parameter for the `select-object` cmdlet. If you want to make the `property` parameter explicit, use the following form of the command:

```
get-childitem |
select-object -property Name, Definition |
sort-object Name |
more
```

The `sort-object` cmdlet and its parameter specify that the objects in the pipeline are to be sorted according to the value of their `Name` property. If you want to review all members of the `sort-object` cmdlet, use the following command:

```
get-command sort-object | get-member
```

Let's move on now to look at some of the key cmdlets that come with PowerShell.

PowerShell supports five cmdlets that allow you to access or manipulate aliases:

- `export-alias`
- `get-alias`
- `import-alias`
- `new-alias`
- `set-alias`

In the following sections, I will describe what each of these cmdlets does and show you examples of how you can use them.

Microsoft, or third-party providers, may in future support additional cmdlets. Also, you can create your own functions or filters. You can use the command

```
get-command *-alias
```

to confirm the supported cmdlets, functions, and filters available on your system for manipulating aliases.

The export-alias Cmdlet

The `export-alias` cmdlet allows you to export a list of aliases to a file. You can export aliases in comma-separated value format or script format. The default format is the comma-separated value format.

To export the current alias list to a text file, use a command like this:

```
export-alias c:\Alias.txt
```

The above command uses an implicit positional `path` parameter. If you want to make that parameter explicit in the command, use this form:

```
export-alias -path c:\Alias.txt
```

You can, for example, open `c:\Alias.txt` in Notepad. Alternatively, you can use PowerShell to display the content, using the `get-content` cmdlet, as follows:

```
get-content c:\Alias.txt
```

Figure 5-8 shows one screen of the content of `c:\Alias.txt`. Notice that the values are in comma-separated format.

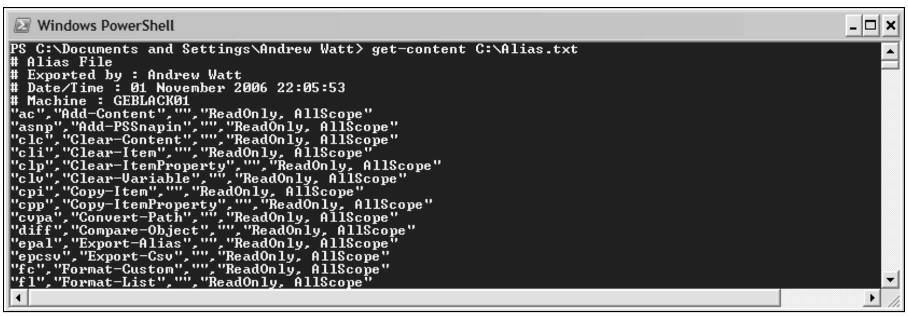


Figure 5-8

The `export-alias` cmdlet takes each of the aliases in the current alias list and writes the values of the relevant members of the objects to a file.

The `get-content` cmdlet allows you to display the content of a file. By default, PowerShell creates two useful aliases for `get-content`: `cat` and `type`.

You can export the alias list as script by using the `–as` parameter. The following command exports the alias list to a script file, `c:\Alias.txt`.

```
export-alias -path C:\Alias.txt -as Script
```

Use this command to open the file in Notepad.

```
notepad C:\Alias.txt
```

Notice in Figure 5-9 that there is a list of commands using the `set-alias` cmdlet.

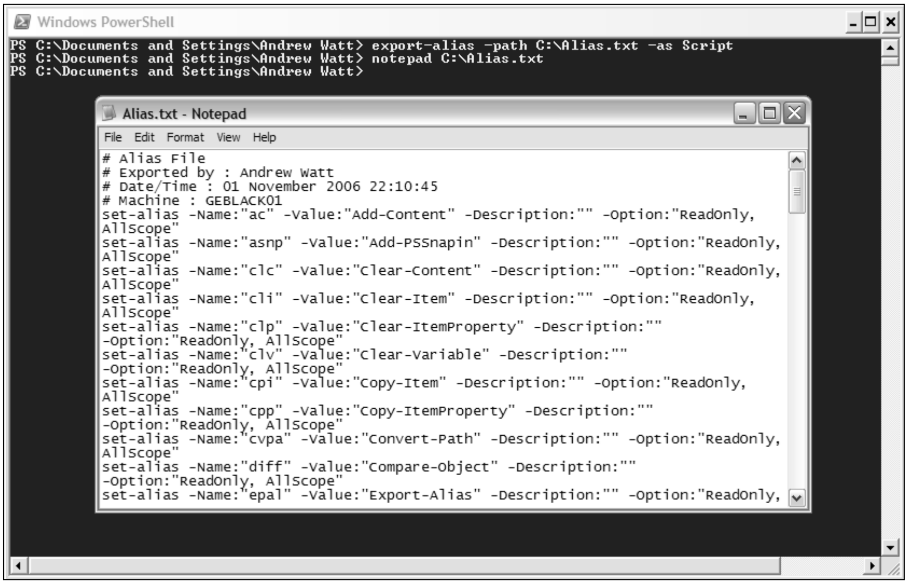


Figure 5-9

If you create multiple alias list files, you can use the `–NoClobber` parameter to test whether or not the file already exists. The following command produces an error message when executed, since the `C:\Alias.txt` file already exists and the `–NoClobber` parameter is specified.

```
export-alias -path C:\Alias.txt -as Script -NoClobber
```

To append an alias list to an existing file, use the `–append` parameter. The following command appends an alias list to the file `C:\OldAlias.txt`:

```
export-alias -path C:\OldAlias.txt -append
```

The get-alias Cmdlet

The `get-alias` cmdlet allows you conveniently to retrieve information about available aliases on a system.

To retrieve a list of all aliases on a system, use any of the following commands:

```
get-alias
```

or:

```
get-alias *
```

or:

```
get-alias -Name *
```

You will likely display more than a screenful of information. Use:

```
get-alias | more
```

or:

```
get-alias |
out-host -paging
```

to display information one screenful of aliases at a time. The two preceding commands do the same. In fact, `more` is an alias that means `out-host -paging`. The `out-host` cmdlet specifies that output is directed to the console. If the `-paging` parameter is present output is displayed one screen at a time. More data is displayed when the user chooses.

You can filter output from the `get-alias` cmdlet to retrieve information relating to specific verbs or nouns by using the `where-object` cmdlet and regular expressions.

To display aliases where the noun of the cmdlet is `process`, use the following command:

```
get-alias |
where-object {$_.Definition -match ".*-process"}
```

Figure 5-10 shows the results. Notice that the cmdlets corresponding to each alias has process as its noun part.

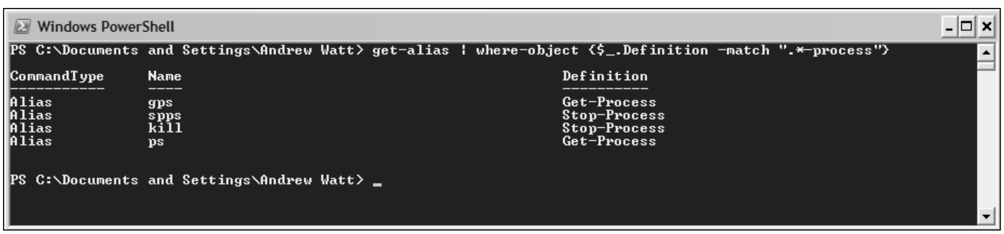


Figure 5-10

The `get-alias` cmdlet without a parameter or value returns objects representing all aliases on the system. The `where-object` cmdlet filters those objects according to the expression contained in the paired curly brackets.

The `$_.Definition` part of the expression refers to the `Definition` property of the current object in the pipeline. It is the `Definition` property that holds the full name of the cmdlet corresponding to each alias. The `-match` parameter specifies that regular expressions are to be used to test the value. The

```
-match ".*-process"
```

part tries to match zero or more characters, followed by a literal hyphen, followed by the literal sequence of characters `process`. Less formally, it will match any verb followed by a hyphen and the noun name `process` (that is, all cmdlets whose noun part is `process`).

The import-alias Cmdlet

The `import-alias` cmdlet retrieves a list of aliases in a file. Assuming that you have aliases defined in a file `d:\UniqueAliases.txt`, you can retrieve those using the following command:

```
import-alias -path d:\UniqueAliases.txt
```

Using the `import-alias` command will generate error messages if an alias is already in use. However, if you have bypassed aliases in a user `profile.ps1` files using

```
PowerShell -noprofile
```

you can then add aliases using `import-alias`.

If you want to use the objects created by the `import-alias` cmdlet in a pipeline, use the `–passthru` parameter.

The new-alias Cmdlet

The `new-alias` cmdlet allows you to create a new alias, associating an alias with the name of a cmdlet or function.

You use the name of the `new-alias` cmdlet, followed by the name of the alias, followed by the name of the cmdlet to which the alias refers. For example, to create a new alias called `getpr` for the `get-process` cmdlet, use the following command:

```
new-alias getpr get-process
```

Or, with the parameter names in full:

```
new-alias –Name getpr –Value get-process
```

Notice that the cmdlet name is supplied as the value of the `–Value` parameter.

You can then use the newly created alias in a command to display running processes. For example, to display processes whose name begins with `sql`, use the following command:

```
getpr sql*
```

Figure 5-11 shows the results on a machine running SQL Server 2005.

Figure 5-11

The `-option` parameter allows you to set optional properties of the alias. The following are the allowed values of the `-option` parameter:

- ❑ `None` — This sets no options.
- ❑ `AllScope` — The alias is available in all scopes.
- ❑ `Constant` — The alias cannot be changed, even by using the `-force` parameter of the `set-alias` cmdlet.
- ❑ `Private` — The alias is available only in the scope specified by the `-scope` parameter.
- ❑ `ReadOnly` — The alias cannot be changed unless you use the `-force` parameter.

The `-scope` parameter specifies the scope of an alias. The allowed values are `global`, `local`, and `script`. The default value is `local`.

The set-alias Cmdlet

The `set-alias` cmdlet allows you to create a new alias or change an existing alias. If used when the target alias doesn't exist, the `set-alias` cmdlet does the same thing as the `new-alias` cmdlet. It maps the name of an alias to the name of a cmdlet.

To create an alias called `date` for the `get-date` cmdlet, use the following command:

```
set-alias date get-date
```

or:

```
set-alias -Name date -Value get-date
```

You can then use the new alias as you would the `get-date` cmdlet. For example, to display the current date and time, simply type the following command:

```
date
```

You can use the alias with the same parameters as the `get-date` cmdlet. For example, to use the `year` parameter and set it to `1966`, use the following command:

```
date -year 1966
```

Figure 5-12 shows the results of executing the preceding two commands.

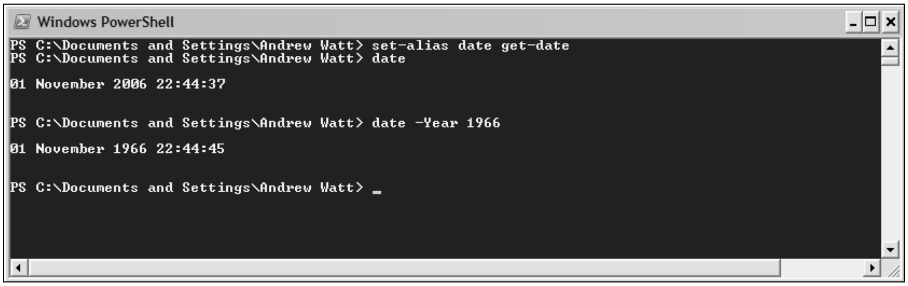


Figure 5-12

Be careful when setting custom aliases using the `set-alias` cmdlet. If the value of the `ErrorActionPreference` preference is set to `Continue` (see the section on preferences later in this chapter), then if you set up an alias `XXX` for a nonexistent cmdlet `get-processes` (remember the noun in the name of a PowerShell cmdlet is always singular):

```
set-alias XXX get-processes
```

no error message is displayed. However, when you later attempt to use the alias:

```
XXX
```

an error message is displayed, as shown in Figure 5-13.

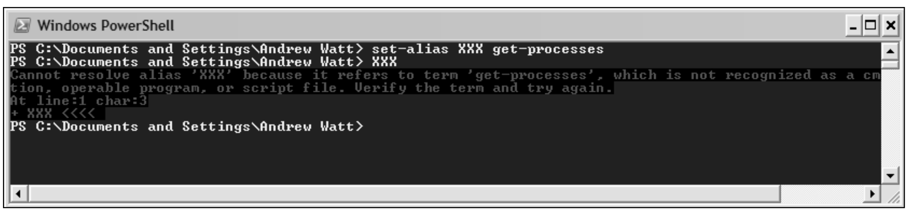


Figure 5-13

The screenshot shows testing of the `XXX` alias immediately after it was created on the command line. However, if you include such a faulty `set-alias` statement in a PowerShell script, the error may not be detected for some time, particularly if the alias was used only inside a conditional statement in the script. This type of possible scenario emphasizes that you need to test all your PowerShell code to make sure that all possible errors are covered.

The Help Alias

You can use the `help` alias in place of the `get-help` command. Strictly speaking, what I am calling the `Help` alias is a function defined in the example `profile.ps1` file shown earlier in this chapter:

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

The `Help` function is defined as being the same as executing the `get-help` cmdlet with one argument, with the result being piped to the `out-host` cmdlet with the `-paging` parameter. The `out-host` cmdlet displays output on the command line. The `-paging` parameter specifies that output is to be displayed one page at a time. Effectively this means that when you type:

```
Help someArgument
```

it is the same as typing:

```
get-help someArgument |
out-host -paging
```

or:

```
get-help someArgument |
more
```

So, to get paged help on the `get-command` cmdlet simply type:

```
help get-command
```

You will see paged help material as shown in Figure 5-14.

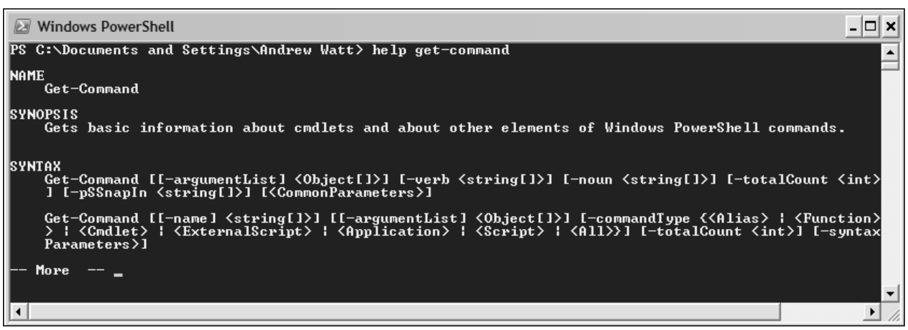


Figure 5-14

You can use the `-detailed` and `-full` parameters with the help function just as you would with the `get-help` cmdlet.

Command Completion

PowerShell offers another cool feature to reduce the number of keystrokes that you have to type: command completion. You begin typing a command or part of a cmdlet and then press the Tab key to have PowerShell complete the cmdlet.

In this example, I show you how command completion works in the registry. PowerShell exposes parts of the registry as the drives `HKCU:` and `HKLM:`. To switch to the `HKLM:` drive, type

```
cd HKLM:
```

at the command prompt. Notice, in Figure 5-15, that the prompt changes to reflect the new selected drive (assuming that the value returned by the `prompt` function includes the result from the `get-location` cmdlet). I explain the `prompt` function in the "Prompts" section later in this chapter.

Figure 5-15

To move down the hierarchy to `Software`, type

```
cd so
```

then press the Tab key. Command completion causes the rest of the word `Software` to be added (as shown in Figure 5-15).

Then type

```
\mi
```

and press the TAB key. The appearance is now as shown in Figure 5-16. PowerShell has completed the word `Microsoft` on the test system. It has also added `HKLM:` following the command `cd`.

When using command completion, you may notice slight inconsistencies in the casing of the text completed, compared to what you typed. Since PowerShell is case-insensitive this doesn't change the behavior of the command, but it can be a little confusing at times to see characters that you have typed replaced by those in a different case.

Figure 5-16

When you press the Return key you navigate to the `Software\Microsoft` folder.

At times, command completion gets in the way rather than helps. For example, if you type a single character, then press Tab, you may end up with the name of a file in the current directory, rather than a cmdlet. Sometimes, you will find that you need to delete all the characters on the line and start again, which tends to defeat the object of using command completion, which ought to increase user convenience.

Command completion seems to work poorly, if at all, for verbs. If, however, you type a verb, then the hyphen, you will often only need to type one or two letters of the noun in a cmdlet name.

Prompts

PowerShell allows you to customize the command prompt. The prompt is defined by the `prompt` function. You can include a function definition for the `prompt` function in a profile file.

The `prompt` function returns a string, which is then displayed to the user at the beginning of each line in the command shell window.

The function definition shown earlier in the example `profile.ps1` file is the default:

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

This definition uses the literal string `PS` followed by a space character, followed by the result returned by the `get-location` cmdlet, followed by a right arrow, followed by a space character. The `get-location` cmdlet, as its name suggests, retrieves the current location. The concatenated string is then displayed at the beginning of each line.

In principle, you can use any PowerShell code to generate a string to use as the prompt. For example, if you want to display the date and time as a prompt, you can use the following definition:

```
function prompt
{
 "" + $(get-date) + " > "
}
```

Figure 5-17 shows the use of the function and the new prompt.

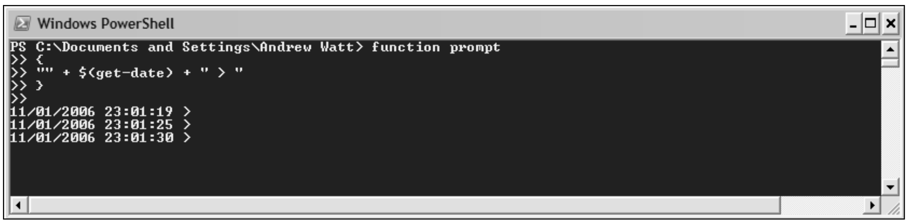


Figure 5-17

Notice that there is an empty string (paired quotation marks with nothing between them) as the first part of the function definition. If you use

```
$(get-date) + " > "
```

in the third line, the desired prompt will not be displayed. See Figure 5-18.

Figure 5-18

Instead, the string `PS` followed by a greater than symbol is displayed, as shown in Figure 5-18. The code in the function definition asks PowerShell to attempt to add a `datetime` object to a string object, which doesn't work, so the fall-back prompt is displayed. Adding an empty string, as shown earlier, resolves that difficulty. The PowerShell parser is then able to work out that you want to create one string for display. The PowerShell parser does a significant amount of automatic type conversion — which makes PowerShell powerful but provides you with considerable flexibility.

To return the prompt to showing the current location, type the following:

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

Preference Variables

You can control some aspects of PowerShell by using preference variables. A preference variable is a variable whose value allows you to express a preference about how PowerShell should behave in a specified situation. For example, the value of the `$ErrorActionPreference` variable allows you to specify how PowerShell should behave if an error is encountered.

To see the preference variables on your system, use all of the following commands:

```
getvariable:*preference*
getvariable:maximum*
getvariable:report*
```

Figure 5-19 shows the results of running the preceding commands.

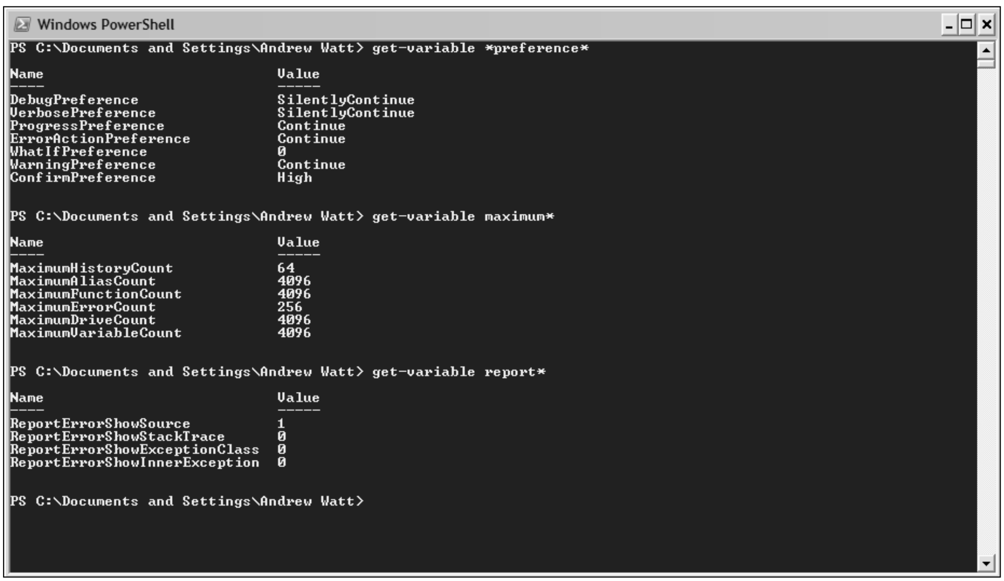


Figure 5-19

As you can see in Figure 5-19, all preference variables and their current values can be displayed using simple commands. If, however, you want to see the value of a specified preference variable, simply type its name on the command line. For example, to see the current value of the `$ErrorActionPreference` variable, simply type

```
$ErrorActionPreference
```

on the command line. The value of the preference variable is then displayed.

Summary

In this chapter, I introduced you to several pieces of Windows PowerShell functionality that you can influence at startup.

A PowerShell snapin is a .NET assembly that contains cmdlets and/or providers whose functionality is in some way related.

Windows PowerShell allows you to use profile files to customize the behavior of the PowerShell console, either for all users on a machine or for an individual user.

Aliases allow you to use familiar commands from other command line environments. You can use a number of aliases that are built-in or create your own.

The `export-alias`, `get-alias`, `import-alias`, `new-alias`, and `set-alias` cmdlets allow you to work with aliases.

6

Parameters

In some situations, you can issue a Windows PowerShell command simply by using the name of a cmdlet or function. This, typically, results in the default behavior of the cmdlet. (If there is a required parameter, you are prompted to provide it.) The cmdlet behaves as if some implicit parameter has been supplied that specifies how the cmdlet is to execute. For example, if you issue the command

```
get-command
```

information about all available commands is displayed in the PowerShell console. The behavior is the same as if you issue either of the following forms of the command:

```
get-command *
```

or:

```
get-command -name *
```

In the two preceding commands, you supply a parameter value (in this case the wildcard *, which matches all command names) that specifies the commands for which information should be displayed.

In some situations, you need to name the parameter before you can supply a value for it. In such situations, the parameter is termed a *named parameter*. In other situations, you don't need to provide a name for some parameters. The position of the parameter value in relation to the position of other unnamed parameter values determines how the PowerShell parser interprets the value that you supply. Parameters for which you can, but don't need to, supply a name are termed *positional parameters*.

Using Parameters

You can use Windows PowerShell commands or functions without specifying any parameters. To retrieve information about all running processes, for example, you can use the `get-process` cmdlet and simply type:

```
get-process
```

Typically, you will see more than a screen of processes listed. The busier the machine, the more difficult it is to see what processes are running. You can use parameters to better focus the results returned by the `get-process` cmdlet. For example, with the `get-process` cmdlet, you can use a `processName` parameter, an `ID` parameter or an `inputObject` parameter to specify how the cmdlet is to execute.

Windows PowerShell parameters are often used by providing a cmdlet name, then a parameter name followed by a space character, then the parameter value. For example, to retrieve information about all running `svchost` processes, type:

```
get-process -processName svchost
```

The cmdlet's name is `get-process`. The parameter's name is `processName` and must be immediately preceded by a minus sign (or hyphen, if you prefer). The parameter value is `svchost`.

If there are no whitespace characters in the value supplied for a parameter, you don't need to supply paired quotation marks or paired apostrophes around the parameter value. So, the command

```
get-process -processName "svchost"
```

is equivalent to the previous command. However, if the parameter value you want to supply contains, for example, a space character, you must enclose the value in paired quotation marks or paired apostrophes.

If the parameter value you supply is a literal value with no contained expression, then paired quotation marks and paired apostrophes are functionally equivalent. Thus,

```
get-process -processName "svchost"
```

which uses paired quotation marks and

```
get-process -processName 'svchost'
```

produce the same result, as you can see in Figure 6-1.

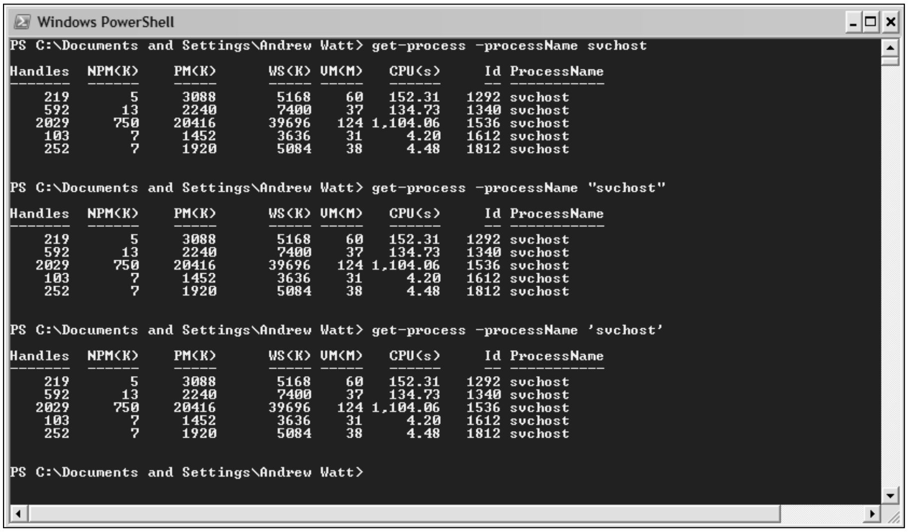


Figure 6-1

When you use paired quotation marks the parameter value is examined for any expressions to be evaluated. Since there are none in this example, it behaves as a simple literal value. When you use paired apostrophes, the content is always treated as if it is a literal value.

> **Windows PowerShell behavior when using paired quotation marks and paired apostrophes is not always the same. When you use paired apostrophes, the value is interpreted literally. When you use paired quotation marks, any expression contained in the paired quotation marks is evaluated.**

In some situations, paired quotation marks will achieve the behavior you want. In others, you need paired apostrophes. For example, suppose that you had assigned the string `svchost` to a variable `$a`:

```
$a = "svchost"
```

If you then use the command:

```
get-process -processName "$a"
```

the value of parameter is arrived at by treating `$a` as an expression, in this case the string `svchost`. However, if you use paired apostrophes:

```
get-process -processName '$a'
```

the content of the paired apostrophes is treated as a literal and the `get-process` cmdlet tries to find processes named `$a`. There are none, so instead of returning information about svchost processes, an error is displayed, as you can see in Figure 6-2.

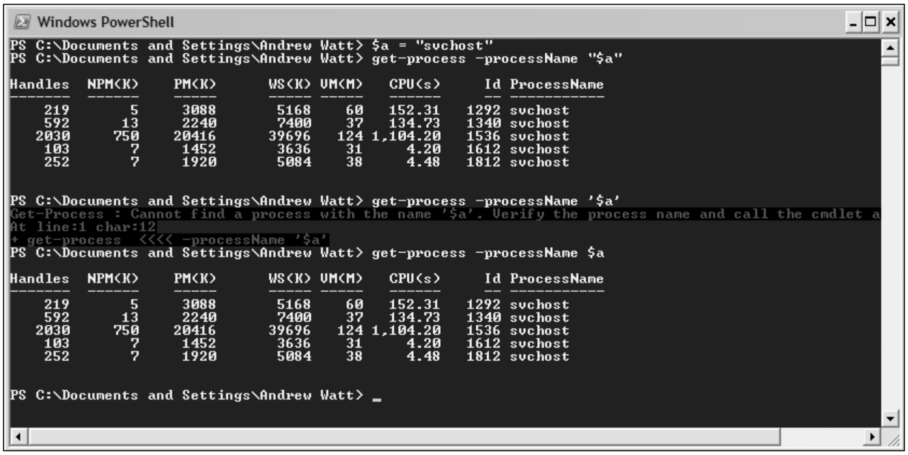


Figure 6-2

If you supply neither paired quotation marks nor paired apostrophes:

```
get-process -processName $a
```

then `$a` is evaluated, in this case to the string `svchost`.

If you want to specify two values for a parameter, separate those values by using a comma and, optionally, one or more space characters. For example to retrieve information about just the `svchost` and `wmiprvse` processes, type the following command:

```
get-process -processname svchost,wmiprvse
```

The Windows PowerShell parser ignores any whitespace between the comma and the second value, as shown here:

```
get-process -processname svchost,  wmiprvse
```

Likewise, you can have one or more space characters before the comma, as shown here:

```
get-process -processname svchost    ,    wmiprvse
```

without changing the behavior of the command.

Each of the preceding three commands retrieves and displays information about processes named `svchost` and `wmiprvse`.

If you want to specify three or more values for a parameter, simply use a comma (plus optional space character(s) to improve readability, if you like) to separate each value. For example, to retrieve information about `notepad`, `svchost`, and `wmiprvse` processes, use this command:

```
get-process -processname notepad, wmiprvse, svchost
```

Parameters that take boolean values don't allow a value to be specified in the normal way. For example, the `paging` parameter for the `out-host` cmdlet doesn't allow a value to be specified by separating the parameter value from the parameter name by a space. You have two options. The first is to simply supply the parameter name. The second is to separate the parameter name from the parameter value with a colon. When you type:

```
get-process | out-host -paging
```

or:

```
get-process | out-host -paging:$true
```

the processes returned are displayed one screenful at a time. Each of the preceding commands does the same thing as:

```
get-process | more
```

That is, it pages the output that reaches the second component in the pipeline. When you specify the `paging` parameter without a value, you are in effect supplying a value `$true`. If you use the `out-host` cmdlet without specifying the `-paging` parameter, you use the default value of the parameter, which is False.

The colon separator is used to specify boolean values for parameters. Notice that if you supply an explicit value, you write `$true` or `$false`, which is the PowerShell syntax for the corresponding boolean values True and False.

Finding Parameters for a Cmdlet

When you are finding your way around Windows PowerShell, you may not be familiar with all the available parameters for a cmdlet. One simple way to address this is to display all of the help file for a cmdlet and pipe the result to `more` or to the `out-host` cmdlet to get paged output. So, you type:

```
get-help get-help |
more
```

or:

```
get-help get-help |
out-host -paging
```

The `get-help` cmdlet displays help information as determined by its accompanying argument and by the presence or not of the `-detailed` or `-full` parameters. You will find information about parameters for the desired cmdlet in the Syntax and Parameters section of the help information.

If you want to avoid unnecessary scrolling, you can display only the information about syntax and parameters. For example, if you want to view the syntax for the `get-service` cmdlet, you can type:

```
(get-help get-service).syntax
```

Figure 6-3 shows the result.

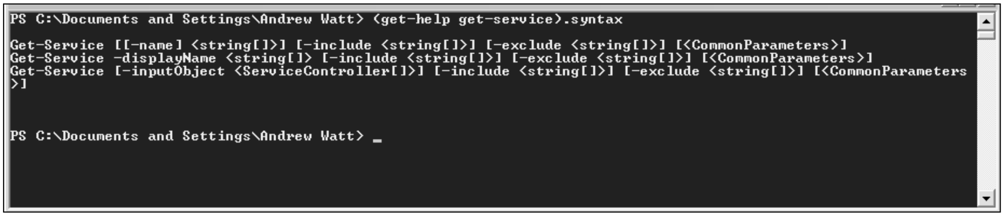


Figure 6-3

By enclosing `(get-help get-service)` in paired parentheses, you are specifying first that Windows PowerShell should produce an object and then that it's the `syntax` property of that object that you want to see displayed. If you typed

```
get-help get-service.syntax
```

an error message would be displayed.

You can use a similar technique to display the detailed parameter information that you would otherwise display by using the `-full` parameter. To display the full parameter information for the `get-service` cmdlet, use the following command:

```
(get-help get-service).parameters
```

As you can see in Figure 6-4, the detailed information about the parameters of the `get-service` cmdlet are displayed.

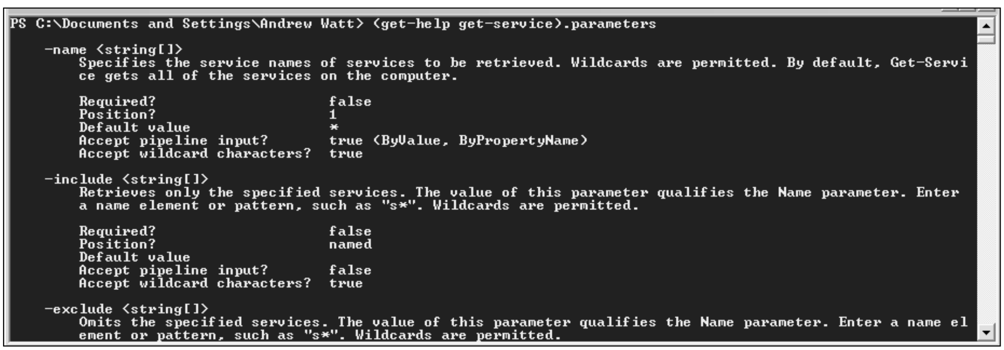


Figure 6-4

Notice in Figure 6-4 that the information about the `-name` parameter indicates that it is a positional parameter. I will discuss positional parameters later in this chapter, in the "Positional Parameters" section. The `-include` parameter is specified as being a named parameter.

As is often the case in Windows PowerShell, there is an alternate syntax to do the same thing. You can use the `-parameter` parameter of the `get-help` cmdlet with a * wildcard. To display detailed information on all the parameters of the `get-service` cmdlet, use either of the following commands:

```
get-help -name get-service -parameter *
```

or:

```
get-help get-service -parameter *
```

Notice that when using the preceding syntax you don't use paired parentheses to enclose any part of the command. The `-name` parameter of the `get-help` cmdlet is positional (explained more fully later in this chapter), so you don't need to supply its name.

It can be useful to combine the two approaches. So, you might, for example, get the names of the `get-help` cmdlet's parameters using this command:

```
(get-help get-help).syntax
```

and look at the details of how to use a selected parameter, in this example the `-name` parameter, using the following command:

```
get-help get-help -parameter name
```

I find it can be a useful way of working if you keep a Windows PowerShell window open primarily to explore cmdlet syntax and another open for using the cmdlets.

There are several other properties in addition to the `parameters` and `syntax` properties. If you want to see all available properties of an object of interest, use this command:

```
get-help get-service |
get-member -memberType properties|
format-table name, definition
```

This will display the names of all the members that are properties.

As you can see in Figure 6-5, among the members are the `details` and the `examples` properties. The `details` property shows you the Name and Synopsis sections of the relevant help information. The `examples` property shows you examples of how you can use the cmdlet of interest.

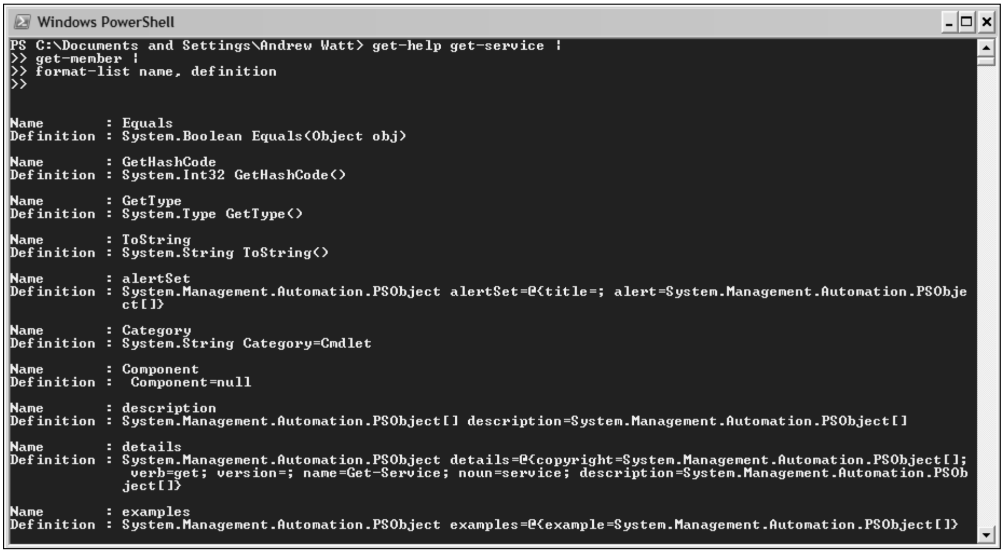


Figure 6-5

Named Parameters

The most common way to use parameters is to specify the name of a parameter and supply its value. For some parameters, termed *named parameters*, this is the only way you can use them. If you don't supply the parameter name, an error is displayed. As I mentioned (and demonstrated) earlier in this chapter, the name of the parameter is immediately preceded by a minus sign (or hyphen, if you prefer). One or more space characters separate the name of the parameter from its value.

The simplest use of named parameters is to supply a literal value for the value of the parameter. So if you are only interested in SQL Server–related services, you can type:

```
get-service -displayName "SQL Server*"
```

The `displayName` parameter is a named parameter, as you can see in the lower part of Figure 6-6.

You can combine multiple literal values separated by commas and optional space characters. However, you may sometimes want the flexibility offered by wildcards, as shown in the preceding example.

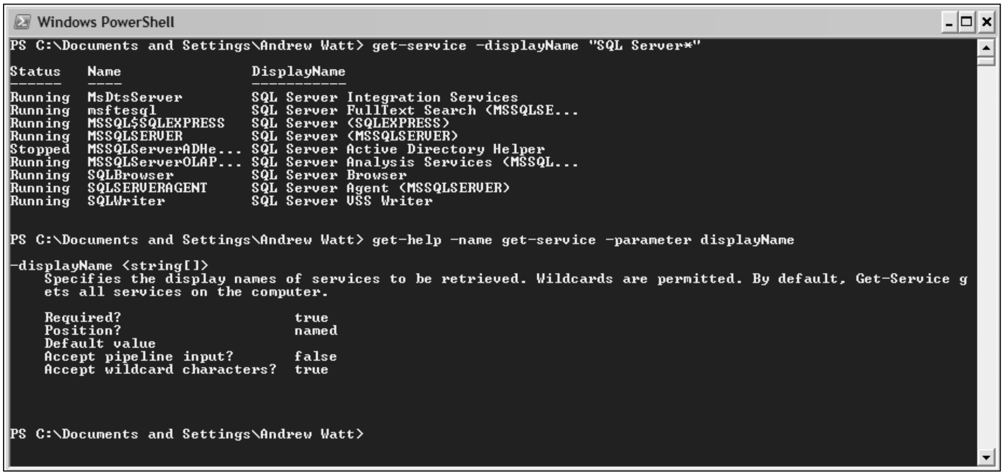


Figure 6-6

Wildcards in Parameter Values

In parameter values, Windows PowerShell supports two wildcard characters. The asterisk matches zero or more characters of any type. The question mark matches a single character.

When you name all parameters that you use in a command, there's no ambiguity about which parameters you intend to use. For example, to retrieve all services beginning with `w` you can use the command:

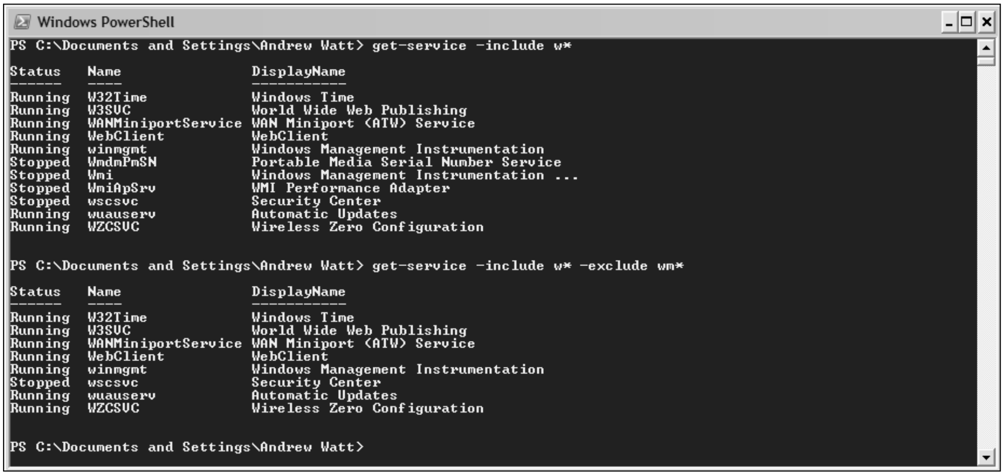
```
get-service -Include w*
```

If you then want to exclude services whose name begins with the characters `wm`, you can add an `exclude` parameter as follows:

```
get-service -Include w* -Exclude wm*
```

As shown in Figure 6-7, the first of those two commands displays all services whose name begins with `w`. The second command displays all of those commands except those that begin with the characters `wm`. Compare the two results, and you'll notice that the `WmdmPmSN`, `Wmi` and `WmiApSrv` services are returned by the first command but not by the second on the machine in question.

The `get-service` cmdlet retrieves objects corresponding to each service on the machine. The `include` parameter specifies that if the name of the service begins with a `w`, the object should be passed along the pipeline. However, any service whose name starts with `wm` is discarded, as specified by the value of the `exclude` parameter.



```
Windows PowerShell
PS C:\Documents and Settings\Andrew Watt> get-service -include w*

Status   Name               DisplayName
------   ----               -----------
Running  W32Time            Windows Time
Running  W3SVC              World Wide Web Publishing
Running  WANMiniportService WAN Miniport (ATW) Service
Running  WebClient          WebClient
Running  winmgmt            Windows Management Instrumentation
Stopped  WmdmPmSN           Portable Media Serial Number Service
Stopped  Wmi                Windows Management Instrumentation ...
Stopped  WmiApSrv           WMI Performance Adapter
Stopped  wscsvc             Security Center
Running  wuauserv           Automatic Updates
Running  WZCSVC             Wireless Zero Configuration

PS C:\Documents and Settings\Andrew Watt> get-service -include w* -exclude wm*

Status   Name               DisplayName
------   ----               -----------
Running  W32Time            Windows Time
Running  W3SVC              World Wide Web Publishing
Running  WANMiniportService WAN Miniport (ATW) Service
Running  WebClient          WebClient
Running  winmgmt            Windows Management Instrumentation
Stopped  wscsvc             Security Center
Running  wuauserv           Automatic Updates
Running  WZCSVC             Wireless Zero Configuration

PS C:\Documents and Settings\Andrew Watt>
```

Figure 6-7

The pipeline in this example is implicit. All objects are piped to the default formatter. The default formatter contains information about a generally useful way to display information about the objects representing each service. I discuss formatting of output in more detail in Chapter 7.

Named parameters can be specified in any order. You can check that by running the command:

```
get-service -exclude wm* -include w*
```

You can also use abbreviated names for the names of each parameter. If you specify an abbreviated name that the Windows PowerShell parser can identify unambiguously, then it is the same as specifying the parameter name in full.

You can retrieve the same objects as specified in the previous example using the command:

```
get-service -inc w* -ex wm*
```

Figure 6-8 shows the information about the specified services.

Notice, too, in Figure 6-8 that if you supply an ambiguous abbreviated parameter name, then an error message is displayed. In this case, the command

```
get-service -in w* -ex wm*
```

is ambiguous. There are two parameters that begin with `in`, the `include` parameter and the `inputObject` parameter.

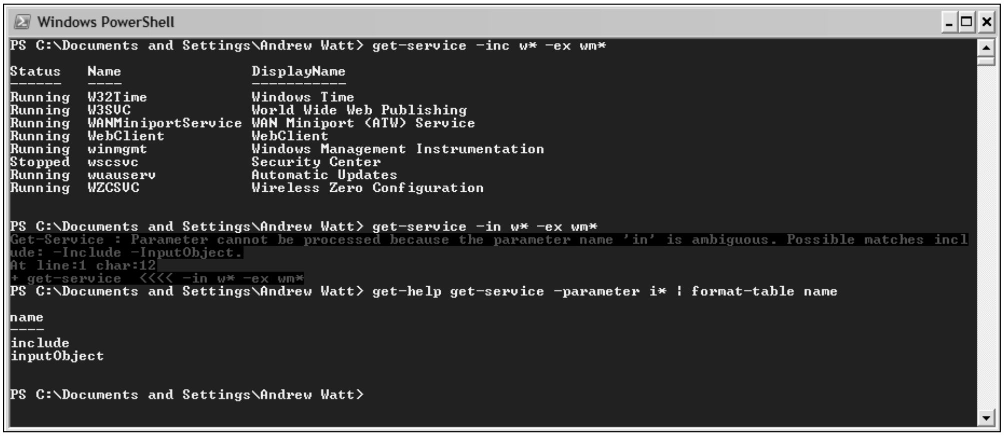


Figure 6-8

When such an ambiguous situation produces an error, you can simply add another letter to the parameter's abbreviated name and try again or use the `get-help` cmdlet to display all relevant available parameter names:

```
get-help get-service -parameter i*
```

and adjust the command accordingly.

To use abbreviated parameter names simply provide enough of the parameter name to enable the Windows PowerShell parser to identify the parameter. To remind yourself of the available parameters, use the techniques described.

Positional Parameters

For selected parameters, Windows PowerShell allows you to omit the parameter name completely. You can simply type the value of the parameter without providing its name. To be able to disambiguate the meaning of parameter values supplied in that way, Windows PowerShell needs to know the position where the parameter is used in relation to any other parameter values whose parameter name has not been specified. These parameters are, therefore, called *positional parameters*.

You can discover positional parameters for a cmdlet by using the `select-object` command. For example, to find the positional parameters for the `get-process` cmdlet, use this command:

```
(get-help get-process).parameters.parameter |
select-object name, position
```

The values in the Position column are displayed so that parameters with a numeric value in that column are displayed first. However, if you want to display only information about the positional parameters, you can add a step to the pipeline that filters objects by using the `where-object` cmdlet:

```
(get-help get-process).parameters.parameter |
where-object {$_.position -ne "named"} |
select-object name, position
```

This returns information about the positional parameters for the `get-process` cmdlet. One parameter, the `name` parameter for the `get-process` cmdlet, is a positional parameter, as you can see in Figure 6-9. It can be used positionally if it is the first parameter (which is not a named parameter) after the cmdlet name, as indicated by its position property value of 1.

In the first form of the command, using `(get-help get-process)` in parentheses returns an object that contains the help for the `get-process` cmdlet. You then use one of the properties of that object, the `parameters` object of which you use the `parameter` property.

The `select-object` cmdlet in the second step of the pipeline selects the name and position of each parameter. If a parameter can be used as a positional parameter, its allowed position is returned. If a parameter can be used only as a named parameter, the value of the `position` property is `named`, indicating that the parameter name must be supplied when it is used.

In the second form of the command using the `where-object` cmdlet the script block in curly brackets uses the `-ne` (not equal) operator to discard objects that have the value `named` for their `position` property. The effect of this is that only objects representing positional parameters are passed to the third step of the pipeline.

Using a positional parameter is very straightforward. You simply use the value of the parameter where you would, for a named parameter, have used the name and value pair.

In this example, you find all processes beginning with `w` using a named parameter then carry out the same task using the parameter as a positional parameter.

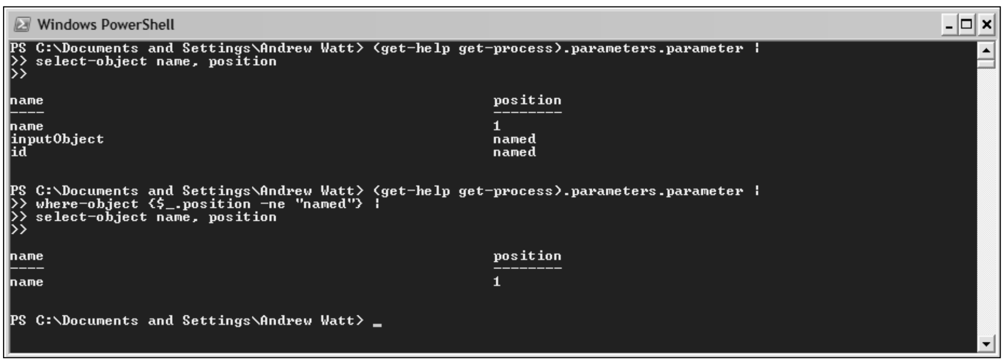


Figure 6-9

The verbose syntax (that is, supplying the name of the parameter) to find all processes beginning with `w` is

```
get-process -Name w*
```

Using a positional parameter, simply type:

```
get-process w*
```

All processes whose process name begins with `w` are displayed, as shown in Figure 6-10.

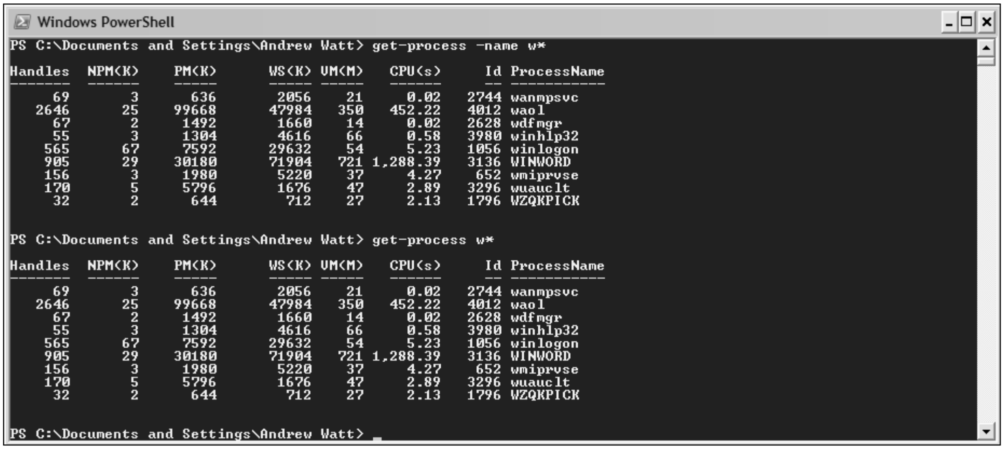


Figure 6-10

The `-name` parameter can be used in position 1 after the cmdlet name. It is the only positional parameter for the `get-process` cmdlet. So, when you type:

```
get-process w*
```

the Windows PowerShell parser knows that the only parameter which can legally be supplied without a parameter name is the `-name` parameter. That must be supplied in position 1, as here. Named parameters are ignored by the parser when determining position. So, the string `w*` is interpreted as being the value of the positional parameter `-name`.

Some cmdlets, for example the `trace-command` cmdlet, have multiple positional parameters. (The `trace-command` cmdlet is used in debugging, which I describe in Chapter 18.) To view the parameters of that cmdlet, type:

```
(Get-Help trace-command).parameters.parameter |
select-object name, position
```

As you can see in Figure 6-11, the `trace-command` cmdlet has multiple positional parameters.

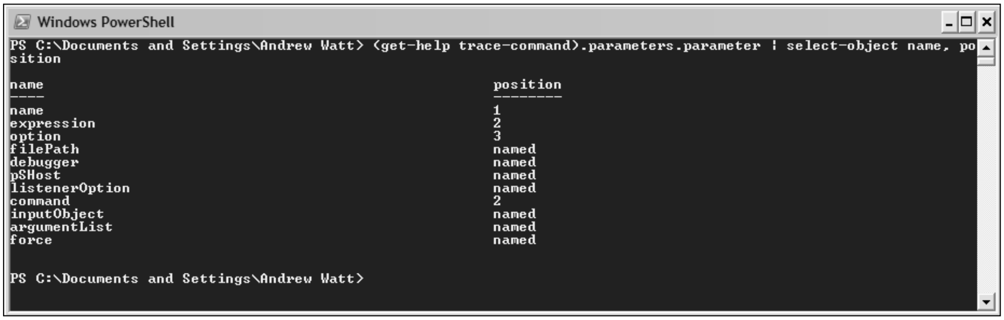


Figure 6-11

Notice that not only does the `trace-command` cmdlet have multiple positional parameters, but, more confusing, two parameters have a value of 2 in the position column.

To understand what happens, take a look at using two positional parameters together. In this example, use the `-name` and `-expression` positional parameters. The following command executes the command `get-process notepad` and displays debugging information. Notice that in the first form the names of the `-name` and `-expression` parameters are used explicitly.

```
trace-command -name metadata,parameterbinding,cmdlet -expression {get-process
notepad} -pshost
```

However, the command works identically if you omit the names of the `-name` and `-expression` parameters.

```
trace-command metadata,parameterbinding,cmdlet {get-process notepad} -pshost
```

The `pshost` parameter is a named parameter that specifies that the output of the trace is to be sent to the Windows PowerShell console (or host). You can move the `-pshost` parameter so that it is before the two positional parameters, and the command behaves as before. As I mentioned earlier, named parameters are ignored when the parser determines the position of a positional parameter. The value intended as the value of the `-name` parameter still comes first, and the value of the `-expression` parameter still comes second, so it all works nicely.

```
trace-command -pshost metadata,parameterbinding,cmdlet {get-process notepad}
```

But why are there two parameters, `-expression` and `-command` in Figure 6-11 that have a position of 2?

To see an overview of the parameters of the `trace-command` cmdlet, execute the following command:

```
(get-help trace-command).syntax
```

Figure 6-12 shows the results.

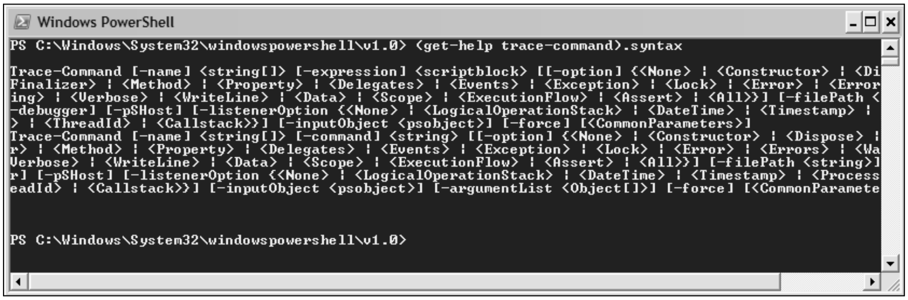


Figure 6-12

It may not be obvious in Figure 6-12, but the parameters of the `trace-command` cmdlet can be used in two *parameter sets*. The first parameter set is this. Notice that it contains the `-name` and `-expression` parameters (in bold).

```
Trace-Command [-name] <string[]> [-expression] <scriptblock> [[-option] {<None> |
<Constructor> | <Dispose> | <
Finalizer> | <Method> | <Property> | <Delegates> | <Events> | <Exception> | <Lock>
| <Error> | <Errors> | <Warn
ing> | <Verbose> | <WriteLine> | <Data> | <Scope> | <ExecutionFlow> | <Assert> |
<All>}] [-filePath <string>] [
-debugger] [-pSHost] [-listenerOption {<None> | <LogicalOperationStack> |
<DateTime> | <Timestamp> | <ProcessId
> |
<ThreadId> | <Callstack>}] [-inputObject <psobject>] [-force] [<CommonParameters>]
```

The second parameter set contains the `-name` and `-command` parameters.

```
Trace-Command [-name] <string[]> [-command] <string> [[-option] {<None> |
<Constructor> | <Dispose> | <Finalize
r> | <Method> | <Property> | <Delegates> | <Events> | <Exception> | <Lock> |
<Error> | <Errors> | <Warning> | <
Verbose> | <WriteLine> | <Data> | <Scope> | <ExecutionFlow> | <Assert> | <All>}] [-
filePath <string>] [-debugge
r] [-pSHost] [-listenerOption {<None> | <LogicalOperationStack> | <DateTime> |
<Timestamp> | <ProcessId> | <ThreadId> |
<Callstack>}] [-inputObject <psobject>] [-argumentList <Object[]>] [-force]
[<CommonParameters>]
```

If you use the first parameter set, then the positional parameter in position 2 is the `-expression` parameter. If you use the second parameter set, the positional parameter in position 2 is the `-command` parameter.

Common Parameters

Windows PowerShell supports six *common parameters*. As the name suggests, these parameters are available generally for use with all cmdlets.

The common parameters are:

- `Debug` — A boolean value that specifies whether or not debugging information is collected. Debugging information is displayed only if the cmdlet supports generation of debugging information.
- `ErrorAction` — Specifies behavior when an error is encountered. The allowed values are Continue (which is the default behavior), Stop, Silently Continue, and Inquire.
- `ErrorVariable` — Specifies the name of a variable that stores error information. The specified variable is populated in addition to `$error`.
- `OutBuffer` — Specifies the number of objects to buffer before calling the next cmdlet in the pipeline.
- `OutVariable` — Specifies a variable to store the output of a command or pipeline.
- `Verbose` — If this parameter is specified, then verbose output is generated, if the cmdlet supports verbose output. If the cmdlet does not support `-verbose` output, then the parameter has no effect.

Each of the common parameters has an abbreviation that you can use in its place, as shown in Table 6-1.

Ubiquitous Parameter	Abbreviation
`-Debug`	`-db`
`-ErrorAction`	`-ea`
`-ErrorVariable`	`-ev`
`-OutputBuffer`	`-ob`
`-OutputVariable`	`-ov`
`-Verbose`	`-vb`

If a cmdlet changes system state two other parameters are available:

- `Confirm` — The user is asked to confirm an action before it is carried out.
- `WhatIf` — The user is shown the actions that the system *would have* taken if the command had been executed. The command does not change the system state.

Using Variables as Parameters

So far I have shown you how to provide literal values as the values for parameters. However, you can also use Windows PowerShell variables as the values for parameters.

I will describe Windows PowerShell variables in more detail in Chapter 10.

A Windows PowerShell variable is named with a dollar sign followed by uppercase or lowercase alphabetic characters, numeric digits, and/or the underscore character.

Do not use the variable `$_` in your code. Windows PowerShell uses that variable as an internal variable in pipelines. Scoping of variables often prevents ambiguity for the Windows PowerShell parser, but I strongly recommend that if you use the underscore character it be combined with alphabetic characters or numeric digits in a variable name.

To use a variable as the value of the parameter, you simply provide the variable name where you would provide a literal value.

In this example, I will show you how to use a variable to supply a parameter value to a cmdlet.

In examples earlier in this chapter, you looked for processes that began with `w`, using the command

```
get-process w*
```

You can achieve the same thing using a variable. First, assign the wildcard `w*` to the variable `$a`:

```
$a = "w*"
```

Then you can supply the value of the variable to the cmdlet:

```
get-process -name $a
```

or, since the `-name` parameter of the `get-process` cmdlet is a positional parameter:

```
get-process w*
```

As you can see in Figure 6-13, processes that begin with `w` are returned.

The Windows PowerShell parser recognizes that a `-name` parameter is being supplied for use with the `get-process` cmdlet. It has access to the values of all in-scope variables. The value of the `$a` variable is retrieved and used as the value of the `-name` parameter. The string `w*` includes a wildcard character, `*`, which matches zero or more characters. Processes that have an initial literal `w` followed by zero or more additional characters are displayed. In other words, processes whose process name begins with `w` are displayed.

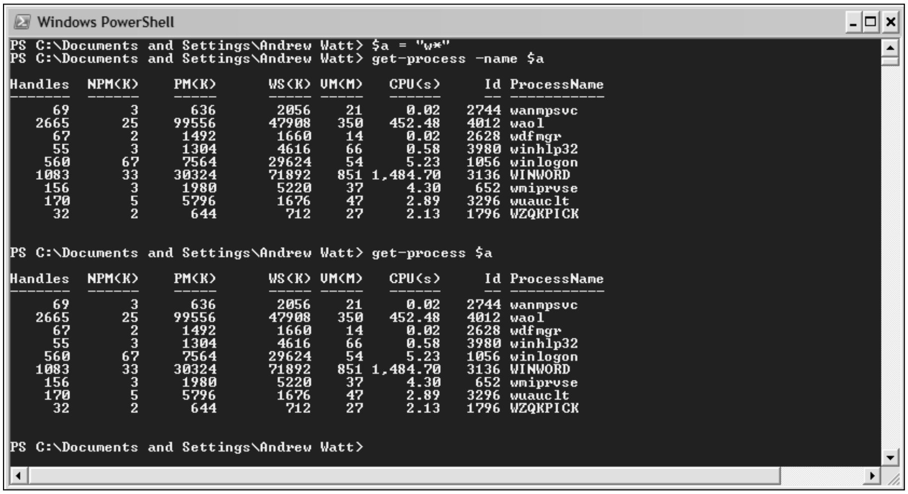


Figure 6-13

You can also write some Windows PowerShell code to get the value for an optional parameter from the user by using the `read-host` cmdlet. If you omit a required parameter, you are prompted by PowerShell to enter a value.

The `read-host` cmdlet accepts a value supplied by the user. You must supply a prompt to be displayed to the user, with the `prompt` parameter. So, to ask a user to supply a value for processes to be searched for, you use this command:

```
read-host -prompt "Enter name of processes to search for"
```

Notice in Figure 6-14 that Windows PowerShell automatically supplies a colon and a space character after the prompt that you provide.

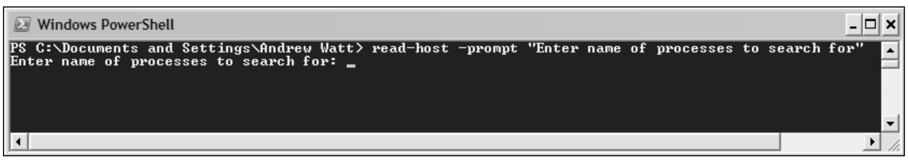


Figure 6-14

Any value supplied is simply echoed back to the console. However, once you assign the user-supplied value to a variable, you can use the value elsewhere in your code. To assign the user-supplied value to the variable `$a`, use this code:

```
$a = read-host -prompt "Enter name of processes to search for"
```

The user will enter the string `w*`, which will find the same processes as in the previous example, assuming that no processes whose name begins with `w` have been stopped or started in the interim.

Confirm that the user-supplied value has been captured by typing:

```
$a
```

which simply echoes the current value of the variable to the console.

Then you can use the user-supplied value to display processes whose name begins with `w`, as shown in Figure 6-15, by using the following command:

```
get-process -name $a
```

as in the previous example.

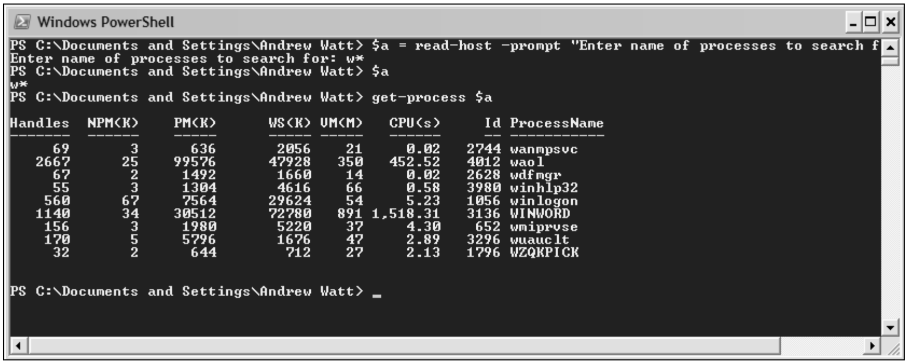


Figure 6-15

The `read-host` cmdlet assigns the value supplied to the user to the variable `$a`.

The Windows PowerShell parser can then use the current value of that variable when it parses the command:

```
get-process -processname $a
```

This technique allows you to get values from the user at runtime when you use Windows PowerShell code in script files. Using scripts provides you with much more flexibility after you have tested your code on the command line. Scripts are introduced in Chapter 10.

Summary

When you use a Windows PowerShell cmdlet, you will often use the cmdlet with one or more parameters. A Windows PowerShell cmdlet's parameter is supplied as a name immediately preceded by a hyphen. The value is separated from the name by one or more whitespace characters. A parameter's value may contain multiple elements separated by commas.

Named parameters are parameters whose name must be supplied. A positional parameter (of which a cmdlet may have none or more than one) can be interpreted by the PowerShell parser without the parameter's name being specified.

7

Filtering and Formatting Output

In this chapter, I cover two techniques that you may find useful to take control of your output. You look at how you can take the potentially enormous amount of information returned from some cmdlets and how to format and filter that information.

Filtering determines whether or not an object is passed on to the next step in a pipeline. You invoke filtering by using some cmdlets and specifying tests that determine what objects to pass along the pipeline. The `where-object` cmdlet is a powerful tool for filtering according to a test specified in a Windows PowerShell expression. You can also use the `select-object` cmdlet to select specified properties to be passed along the pipeline.

Formatting is concerned with the display of information, both in general and in determining where the objects are supplied to the final step in a pipeline. In many of the pipelines you have seen so far, there has been an invisible final step that uses the *default formatter* to define how the results of a command or pipeline are displayed. However, the default formatter doesn't always format the output in the way you need. Windows PowerShell provides two cmdlets, `format-table` and `format-list`, which allow you to take more control of the display of the information in objects that emerge from earlier steps in the pipeline.

Using the where-object Cmdlet

The `where-object` cmdlet filters the objects presented to it. Most commonly, when you use the `where-object` cmdlet on the command line, the objects it filters will come from an earlier step in a pipeline.

In a standard install, you are likely to have two aliases available to use in place of the full form of `where-object`: `where` and ?. To find the aliases available for the `where-object` cmdlet on your system use this command:

```
get-childitem alias:\ |
where-object -filterScript{$_.Definition -match "where"}
```

or:

```
get-childitem alias:\ |
where-object {$_.Definition -match "where"}
```

The `-filterScript` parameter is used to filter objects. As the name of the parameter suggests, its value is a script. The script is enclosed in paired curly braces. The `-filterScript` parameter is a positional parameter in position 1.

Simple Filtering

The argument to the `where-object` cmdlet is a Windows PowerShell expression that is the value of the `-filterScript` parameter, and it returns a boolean value. The expression is contained in paired curly brackets and uses a number of operators, which I describe later in this section.

You might want to find out which services on a machine are running. Of course, you can do that using the Services Microsoft Management Console snapin, but you can also do it easily from the Windows PowerShell command line. The `get-service` cmdlet finds all services installed on a machine. This example uses the `where-object` cmdlet to display information only about running services.

If you are unsure about what members are available on the objects returned by the `get-service` cmdlet, use

```
get-service |
get-member
```

to display all the public members.

For the purposes of this example, it is the `status` property that is of interest, since the value of the `status` property shows whether a service is running or stopped. Sometimes you may not be familiar with the values allowed for a property or that apply in a particular setting. In that situation, one way to get a handle on the available values is to use the property with the `select-object` cmdlet. For example, to find out what values of the `status` property apply to objects on a machine, type this command and then scan the displayed results:

```
get-service |
select-object name, status
```

You will see output similar to Figure 7-1, which shows part of the output on one machine.

Figure 7-1

An alternate approach to do a similar thing is to use the `format-table` cmdlet to display the columns of interest, as in the following command:

```
get-service |
format-table name, status
```

When you have multiple screens of unsorted information, it can be hard to scan each line to check which values are present. You can use the `group-object` cmdlet to find the values for the `status` property. Use the following code to do that:

```
get-service |
select-object name, status |
group-object status
```

Figure 7-2 shows the result. Instead of having to scan the values for the `status` property for over 100 objects, Windows PowerShell does the work for you.

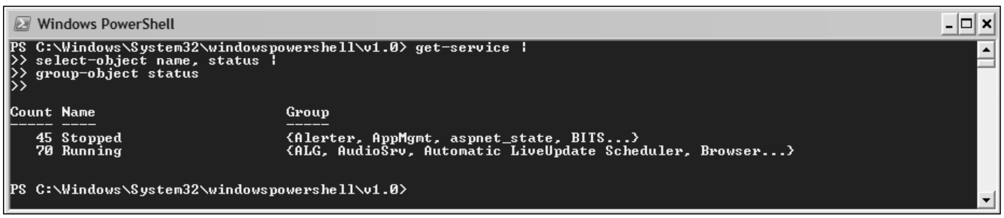


Figure 7-2

As you can easily see, only two values are in use for the `status` property: `stopped` and `running`. In addition, some services can be paused, in which case the `status` property has a value of `paused`. So to find out which services are running on the machine, you need to find services where the value of the `status` property is `running`. The following code does that:

```
get-service |
where-object {$_.status -eq "running"}
```

Figure 7-3 displays part of the results from the command.

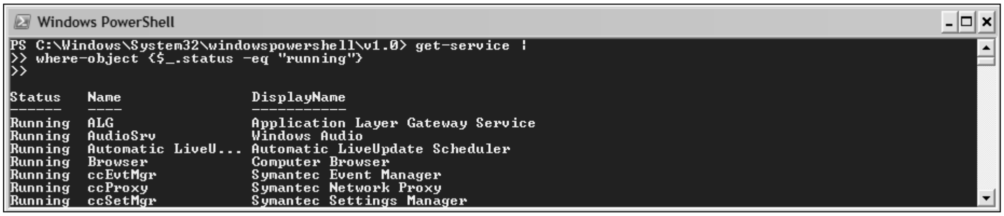


Figure 7-3

If you prefer to use aliases when writing the preceding command you can use:

```
gsv |
where {$_.status -eq "running"}
```

or:

```
gsv |
? {$_.status -eq "running"}
```

Having the status column at the left of the display may not be ideal for you. Later in the chapter I will show you how you can improve the way Windows PowerShell displays results.

The `get-service` cmdlet, when used with no parameters, returns objects representing all services on the machine.

The `where-object` step in the pipeline filters the objects representing services according to the value of their `status` property. The `$_` variable is a special variable that essentially means "this object." In other words, the `$_` variable successively refers to each of the objects passed into the pipeline from the `get-service` cmdlet. The `-eq` operator tests each object's `status` property to see if the service is running.

> **In Windows PowerShell, you cannot use `=` to test for equality. The `=` operator is the assignment operator. Use the `-eq` operator instead.**

Using Multiple Tests

In more complex pipelines, you may want to filter on more than one criterion. You can use multiple filters based on the `where-object` cmdlet in the same pipeline.

In this example, I show you two ways to combine two filter criteria in a pipeline. The desired services are running services that include the sequence of characters `sql` in their name. This allows you to see which SQL Server services are running. Choose another service if you don't have SQL Server installed.

> **Testing for "`sql`" in a service name isn't entirely specific, since some other programs, for example mySQL, would also match the specified criteria. You need to know your system to decide whether an approach such as this is sufficiently specific.**

One technique is simply to use the `where-object` cmdlet in two steps of a pipeline. To do that, use this command:

```
get-service |
where-object {$_.status -eq "running"} |
where-object {$_.name -match ".*sql.*"}
```

Figure 7-4 shows the results on a development machine where SQL Server 20050 and Analysis Services are running.

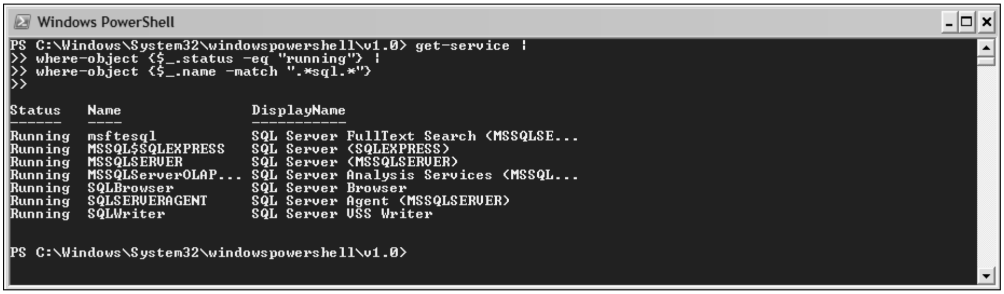


Figure 7-4

A second approach is to combine two filter criteria (or more if you want) using the `-and` operator. The following command applies both filter criteria using a single pipeline step with the `where-object` cmdlet:

```
get-service |
where-object {$_.status -eq "running" -and $_.name -match ".*sql.*"}
```

Figure 7-5 shows the results on the same machine as the previous code. The services that you see on a machine depend, for example, on which SQL Server components you installed on it.

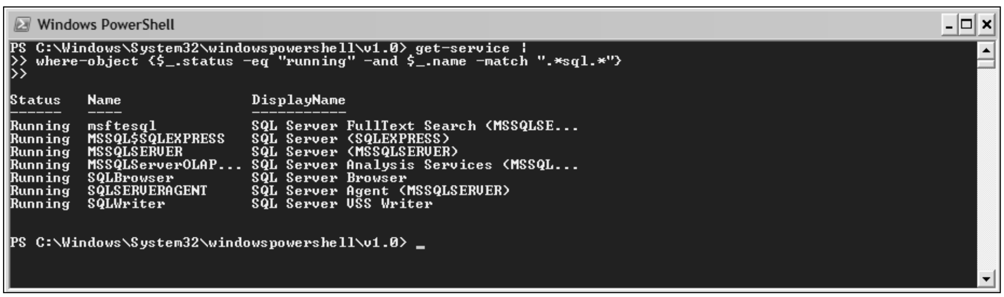


Figure 7-5

First, look at the approach that uses the `where-object` cmdlet twice in a three-step pipeline.

```
get-service |
where-object {$_.status -eq "running"} |
where-object {$_.name -match ".*sql.*"}
```

The get-service cmdlet returns objects representing all services on the machine. The results are piped to the first where-object step, where-object {$_.status -eq "running"}, which passes objects where the value of the status property is running to the next step of the pipeline. All those objects passed to the third step of the pipeline represent running services, so the next use of the where-object cmdlet, where-object {$_.name -match ".*sql.*"}, filters the objects that represent running services so that only those services whose name includes the character sequence sql are passed to the default formatter.

An alternative approach uses two conditions with a single where-object clause, as follows:

```
get-service |
where-object {$_.status -eq "running" -and $_.name -match ".*sql.*"}
```

This uses the get-service cmdlet to pass all services to the second step of the pipeline.

The where-object cmdlet applies two tests when filtering objects. The -and operator specifies that an object must satisfy two tests before it is passed along the pipeline First, as specified by the test $_.status -eq "running", the value of the status property must be running (in other words, the service is running). Those objects where that test is satisfied must also include the character sequence sql in the value of their name property, as specified by $_.name -match ".*sql.*". The -match operator uses regular expressions when matching a name. The .* (a dot followed by an asterisk) matches zero or more characters. The literal character sequence, sql, specifies that those three characters must occur in sequence. Finally, the pattern .* (a dot followed by an asterisk) specifies that the literal sequence is followed by zero or more characters of any type. In other words, the name must include the character sequence sql in any position.

I have shown you how to combine two filters using the where-object cmdlet. As with many situations in PowerShell, there are other approaches. For example, the command

```
get-service *sql* |
where-object {$_.status -eq "running}
```

combines a filter in the value of the positional –name parameter of the get-service cmdlet with a pipeline step that uses the where-object cmdlet with a single test. In day-to-day use this approach is likely to be the best.

The technique to combine two tests using the where-object cmdlet can be adapted to find running (or stopped) services which meet any other criterion of interest to you.

Using Parameters to where-object

The where-object cmdlet can take two parameters, -filterScript and -inputObject.

The -filterScript parameter is a positional parameter (as described in Chapter 6). The earlier examples in this chapter that use the where-object cmdlet use the value of filterScript parameter positionally. You can also express it explicitly.

Suppose that you want to find PowerShell processes that have a handle count greater than 500 and have CPU usage of greater than 5 seconds. You can find those processes by using the `where-object` cmdlet, as in the following command:

```
get-process powershell |
where-object {$_.Handles -gt 500 -and $_.CPU -gt 5}
```

or:

```
get-process powershell |
where-object -filterScript {$_.Handles -gt 500 -and $_.CPU -gt 5}
```

The most common source of objects for filtering by the `where-object` cmdlet is an earlier step in a pipeline. However, the `-inputObject` parameter allows you to use the `where-object` cmdlet to filter Windows PowerShell variables.

Suppose that you had assigned numeric values to three variables `$a`, `$b`, and `$c`:

```
$a = 10
$b = 20
$c = 30
```

You could use the `-inputObject` parameter to test whether the value of each variable was or was not greater than 15. There are easier ways to test this in Windows PowerShell, but the following commands provide an example of how you might use the `-inputObject` parameter. If you use the `-inputObject` parameter in this way, you might test each variable in a collection.

```
where-object -inputObject $a -filterScript {$_ -gt 15}
where-object -inputObject $b -filterScript {$_ -gt 15}
where-object -inputObject $c -filterScript {$_ -gt 15}
```

Figure 7-6 shows the result of executing the preceding commands. Notice that nothing is displayed when the first command is executed, indicating that the value of `$a` is not greater than 15 (which you would expect, since 10 is less than 15).

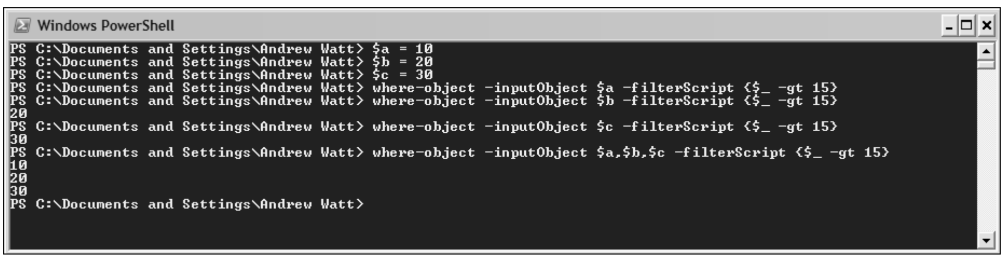


Figure 7-6

At the time of writing, using multiple comma-separated values of the `-inputObject` parameter does not produce the expected results, as you can also see in Figure 7-6:

```
where-object -inputObject $a,$b,$c -filterScript {$_ -gt 15}
```

The where-object Operators

The operators you use in the script block that forms the value of the `-filterScript` parameter are boolean operators. If the boolean operator returns `$true` for an object then the object is passed on for further processing or display. If the boolean operator returns `$false`, then the object is discarded and is unavailable for further processing or displaying. The following table shows the operators that you can use with the `where-object` cmdlet.

Operator	What it does
`-eq`	Tests whether two values are equal.
`-neq`	Tests whether two values are not equal.
`-gt`	Tests whether a first value is greater than a second value.
`-ge`	Tests whether a first value is greater than or equal to a second value.
`-lt`	Tests whether a first value is less than a second value.
`-le`	Tests whether a first value is less then or equal to a second value.
`-like`	Tests whether two values are alike. One value is a string. The other value includes one or more wildcards.
`-notlike`	Same as `-like` buts tests for unlikeness.
`-match`	Tests for a match between a string and a regular expression pattern.
`-notmatch`	Same as `-match` but tests for the absence of a match between a string and a regular expression pattern.

When comparing strings, the `-eq` operator and other comparison operators make the comparison case-insensitively by default. If you wish to make case-sensitive comparisons, add a "`c`" to the beginning of the operator name, viz `-ceq`, `-clt`, and so on. If you wish to make explicit case-insensitive comparisons, add an "i" to the beginning of the operator name, viz `-ieq`, `-ilt`, and so on.

Using the select-object Cmdlet

The `select-object` cmdlet lets you select specified properties of an object or set of objects. In addition, you can use the `select-object` cmdlet to select unique objects from an array of objects or to select a specified number of objects from the beginning or end of an array of objects. The `select-object` has the following parameters in addition to the common parameters listed in Chapter 6:

- `property` — Specifies properties of interest
- `excludeProperty` — Specifies properties to be excluded
- `expandProperty` — Specifies a property to be selected and, if that property is an array, specifies that each value in the array should be selected

- `first` — Specifies a number of values at the beginning of an array that are to be selected
- `last` — Specifies a number of values at the end of an array that are to be selected
- `unique` — Specifies that only unique values are to be selected
- `inputObject` — Specifies an input object, if the input objects are not supplied by the preceding step of a pipeline

I demonstrate how you can use several of these parameters in the sections that follow.

Selecting Properties

You can use the `select-object` with the `property` parameter to select specified properties of an object. One use is to select properties for display. The `property` parameter is a positional parameter, so you can omit the parameter name if you prefer.

Suppose that you want to display the process name and handle count of running processes.

If you select processes using

```
get-process
```

objects representing running processes are returned, then passed to the default formatter. As you can see in Figure 7-7, several columns of information are displayed by default. In this example, you want to see only part of that information.

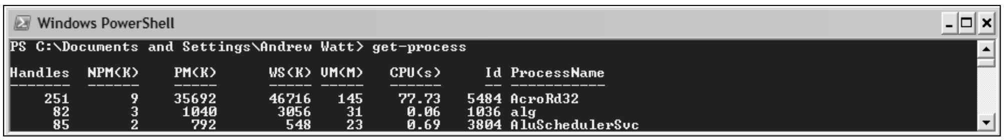


Figure 7-7

To take control of the objects passed to the default formatter so that, for example, only process name and handle count are displayed, use the command

```
get-process |
select-object processname, handlecount
```

to select the process name and handle count of running processes. The result should be similar to Figure 7-8.

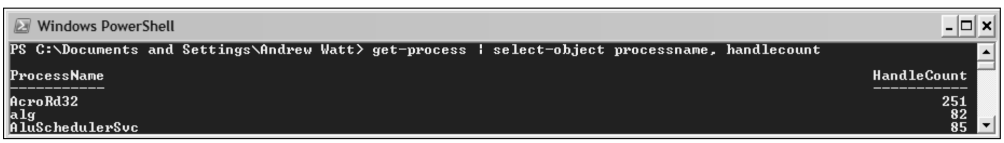


Figure 7-8

The get-process cmdlet in the first step of the pipeline passes objects representing all running processes. The second step of the pipeline is equivalent to

```
select-object -property processname, handlecount
```

so only objects representing the specified properties are passed to the default formatter. The result is that only two columns of data are displayed, in the order specified in the list of values of the property parameter.

You can adapt this technique to display desired properties by using a comma-separated list of the properties that you want to see. The properties are displayed onscreen in the order specified in the comma-separated list.

There are other ways to achieve similar results in Windows PowerShell. For example, you can use the format-table cmdlet (which I describe later in this chapter), as follows:

```
get-process |
format-table processname, handlecount
```

Expanding Properties

You can use the expandProperty parameter of select-object to display additional information about an object or its properties.

In this example, the information associated with the processname property of a process is expanded using the expand parameter.

If you use the following command to display information about processes whose name begins with wmi, as you can see in Figure 7-9 only the name of the process is displayed.

```
get-process wmi* |
select-object processname |
format-list |
more
```

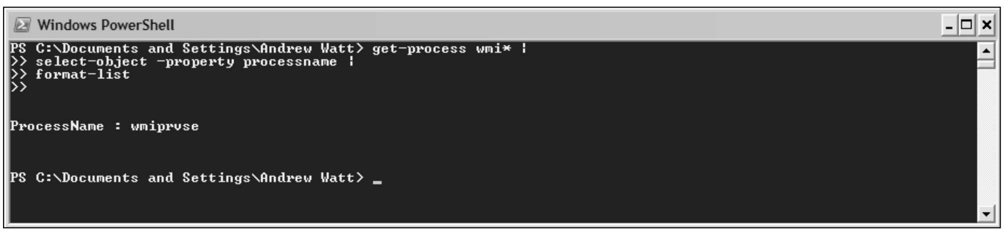


Figure 7-9

However, you can access much more information about these processes from the Windows PowerShell command line. To see information about the modules associated with processes whose name begins with wmi, type the following command. I chose those processes to keep the running time of the command to acceptable limits. If you attempt to display this information for all processes the command may take some time to complete. (If you don't have WMI installed, substitute a process name where you have multiple processes of the same name running on your machine.)

```
get-process wmi* |
select-object processname -expandProperty modules |
format-list |
more
```

Figure 7-10 shows the first screen of results. Notice the substantial amount of additional information that is available including, in this example, information about the path to the file and the version of the file.

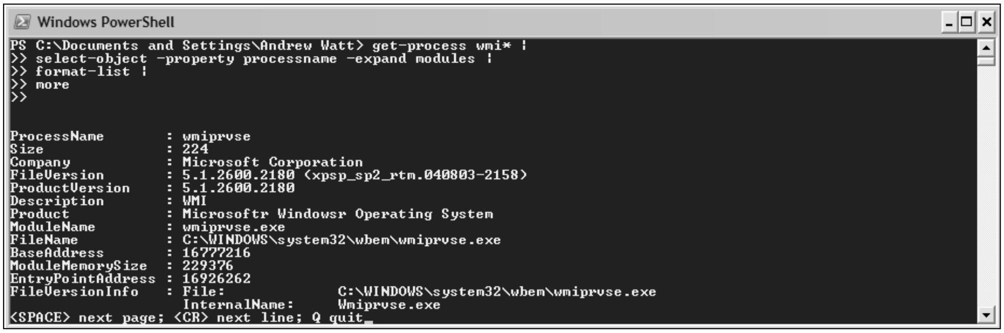


Figure 7-10

The first step of the pipeline, `get-process wmi*`, returns objects representing all processes whose names begin with `wmi`. The `get-process` cmdlet has a positional parameter process name, so you don't need to provide the name of the parameter.

The second step uses the `select-object` cmdlet to expand information through the use of the `expandProperty` parameter. The `format-list` cmdlet simply specifies that the information is to be displayed onscreen as a list.

Selecting Unique Values

The `-unique` parameter allows you to use the `select-object` cmdlet to select only unique values. Suppose that you have a list of values, and you want to find which values are present.

The data is as follows:

1,2,3,1,4,3,4,5,2,7,3,1,2

To find the unique values in this list of values, use the following command:

```
1,2,3,1,4,3,4,5,2,7,3,1,2 |
select-object -unique
```

Figure 7-11 shows the result of executing the preceding command. With extensive data sets, using the `-unique` parameter provides a quick and easy way to see which values are present in a data set.

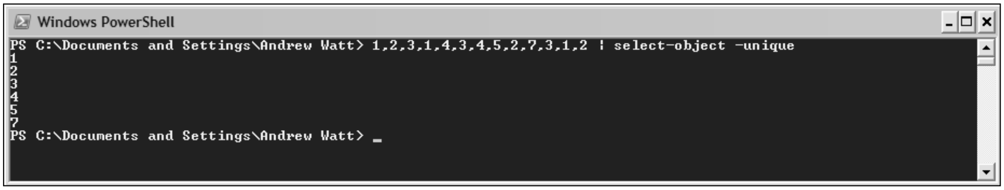


Figure 7-11

You will often want to see sorted data. Simply add a pipeline step that includes the `sort-object` cmdlet.

```
1,2,3,1,4,3,4,5,2,7,3,1,2 |
select-object -unique |
sort-object
```

You can use this approach, for example, to find which processes are running on a machine, irrespective of whether multiple instances of a process are running. The following command displays a list sorted alphabetically by process name:

```
get-process |
select-object -unique |
sort-object
```

You can also use the `-unique` parameter in combination with the `-first` and `-last` parameters, as I show you in the next section.

First and Last

The `-first` parameter allows you select a specified number of values at the beginning of an array of values. The `-last` parameter allows you to select a specified number of values at the end of an array of values.

Suppose that you had the following data set and assigned it to the variable `$a`:

```
$a = 1,8,3,5,9,10,22,1,7,8,9,3,2,11,19,3,8,8,2,4,3,5
```

You can find the first five values by using the following command:

```
$a |
select-object -first 5
```

You pipe the array of values contained in `$a` to the `select-object` cmdlet. The first five values in the array are selected.

As you can see in Figure 7-12, this displays the first five values in the array.

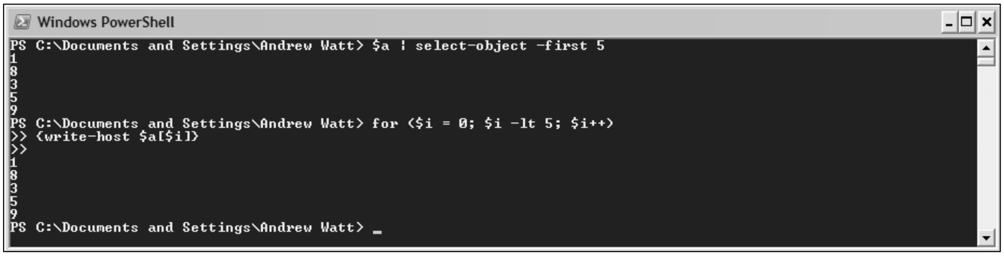


Figure 7-12

Using the `-first` parameter of the `select-object` parameter is easier than displaying the first five values using the Windows PowerShell `for` statement, as follows:

```
for ($i = 0; $i -lt 5; $i++)
 {write-host $a[$i]}
```

Arrays in Windows PowerShell are numbered from 0. by the way. I describe them in more detail in Chapter 11.

Similarly, you can select the last five values in the array by using the following command:

```
$a |
select-object -last 5
```

The `-first` and `-last` parameters are particularly useful when you want to find, say, the smallest five and largest five values in a data set. To use the `select-object` to do that, you need to sort the data first.

To find the five smallest values in `$a`, use this command:

```
$a |
sort-object |
select-object -first 5
```

Similarly, to find the five largest values in `$a`, use this command:

```
$a |
sort-object |
select-object -last 5
```

Figure 7-13 shows the results of executing the preceding commands.

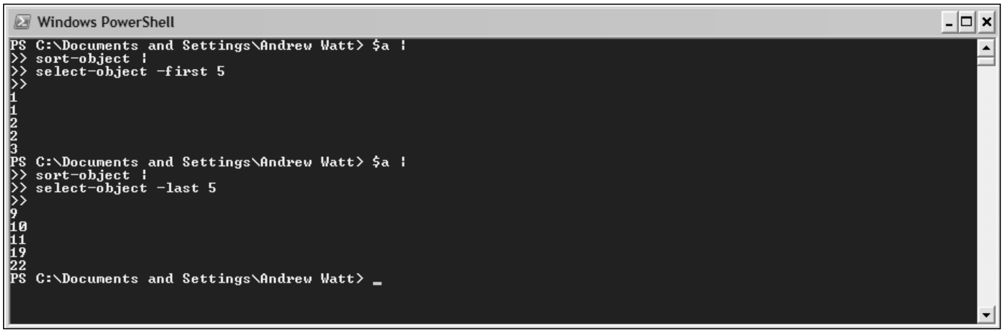


Figure 7-13

The preceding approach allows you to find, for example, processes with the smallest or largest handle counts. The following command finds the five processes with the largest handle count.

```
get-process |
sort-object -descending handlecount |
select-object -first 5
```

The following command uses the `-last` parameter to find the five running processes with the smallest handle count:

```
get-process |
sort-object -descending handlecount |
select-object -last 5
```

You can combine the `-unique` parameter with the `-first` and `-last` parameters. This finds the unique values in the number of values specified by the `-first` or `-last` parameters. For example, to find the unique values in the five smallest values in `$a` use this command:

```
$a |
sort-object |
select-object -first 5 -unique
```

Figure 7-14 shows the preceding command executed without and with the `-unique` parameter. Notice that when it is executed without the `-unique` parameter the values returned are 1,1,2,2,3. There are duplicates for the values 1 and 2. When the `-unique` parameter is specified, the duplicate values are removed.

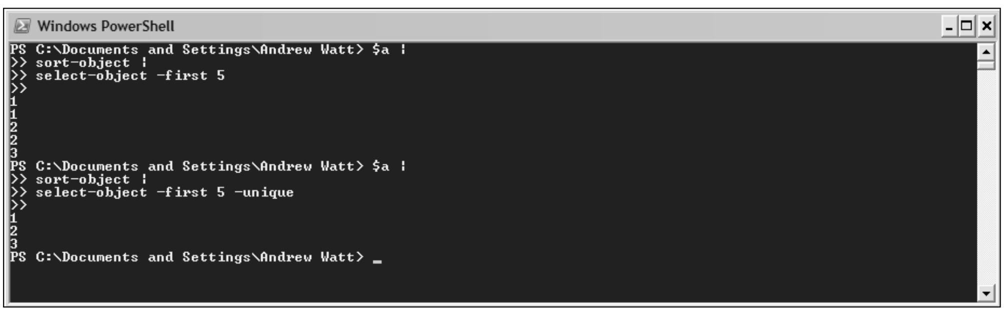


Figure 7-14

Default Formatting

In pipelines that appear to have a single step, such as:

```
get-service
```

there is, in fact, an implicit final step in the pipeline, the default formatter. The visible step passes objects (in this case representing services) to a default formatter.

The default formatter for each cmdlet displays information that Microsoft perceives might be generally useful. Implicitly, the output is piped to the `out-default` cmdlet, which, in turn, pipes the output to the default formatter. The default formatter then displays output.

The file `C:\WINDOWS\system32\windowspowershell\v1.0\powershellcore.format.ps1xml` contains extensive information explaining how information about different types of objects is to be displayed (assuming that you installed Windows on drive C:). Figure 7-15 shows part of a copy of that file displayed in Internet Explorer. The elements refer to `Microsoft.PowerShell.Commands.GroupInfo` objects. Notice the presence of `PropertyName` elements, which are child elements of `TableColumnItem` elements. Three values are contained in those elements – `Count`, `Name`, and `Group`.

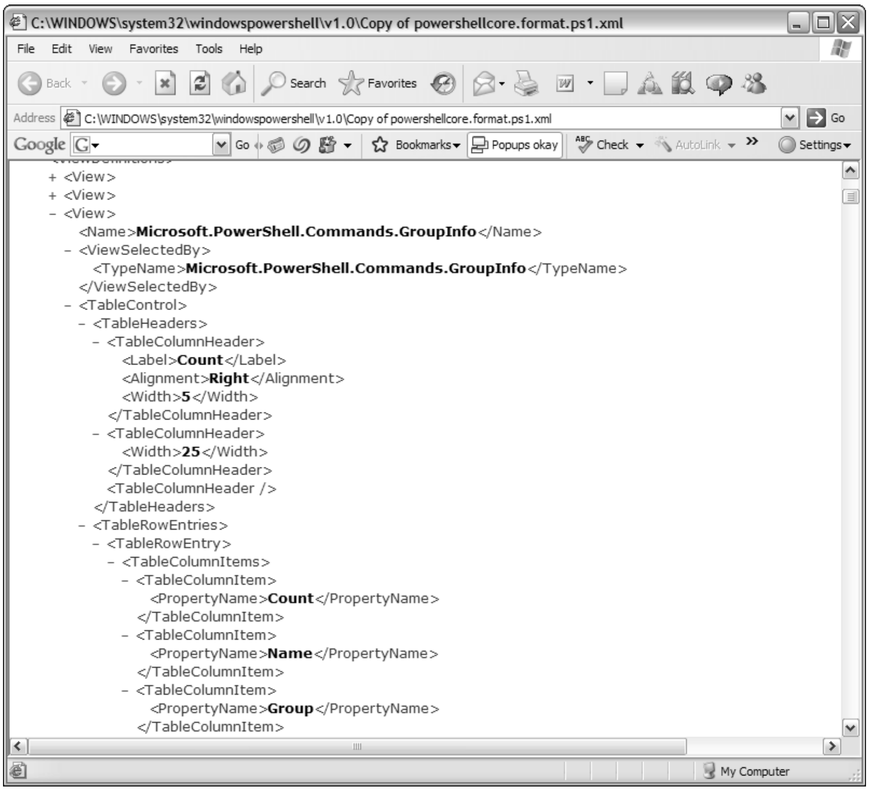


Figure 7-15

Execute the following command to demonstrate the default format from a pipeline using the `group-object` cmdlet as its last explicit step:

```
get-command |
group-object verb
```

Notice in Figure 7-16 that the default formatting for output from the `group-object` cmdlet produces Count, Name and Group columns, corresponding to the `TableColumnItem` elements in the `powershell core.format.ps1xml` file.

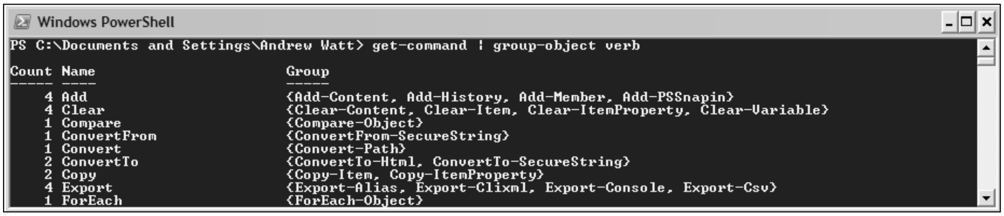


Figure 7-16

You can see in the upper part of Figure 7-17 that the objects produced from the `group-object` cmdlet are `Microsoft.PowerShell.commands.GroupInfo` objects. This confirms that the information displayed in Figure 7-15 applies to the objects output from `group-object` cmdlets.

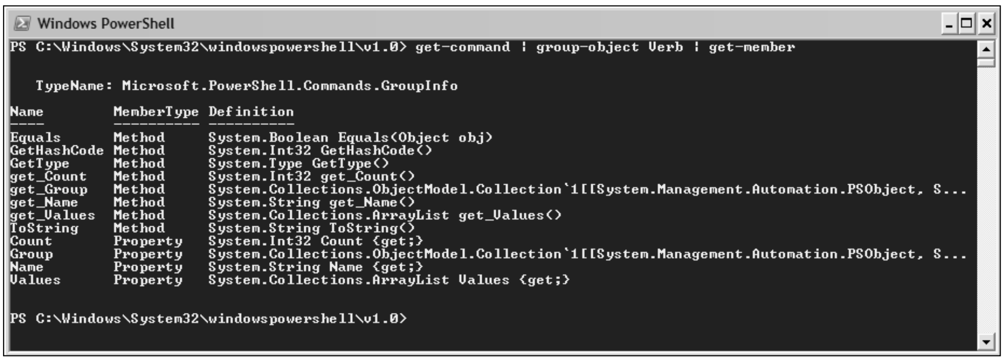


Figure 7-17

However, as I showed you earlier in this chapter, you can use the `select-object` cmdlet as one way to make your own choices about which information to pass to the default formatter. If you specify properties using the `select-object` cmdlet, then the default formatter will display the information in columns corresponding to those objects you specified in the `property` parameter of `select-object` and in the order you specified.

The `powershellcore.format.ps1xml` file also contains information about what should be displayed when you display output as a list. Figure 7-18 shows information in `powershellcore.format.ps1xml` relating to `Microsoft.Management.Automation.CmdletInfo` objects. Notice the `PropertyItem` elements, which are child elements of `ListItem` elements.

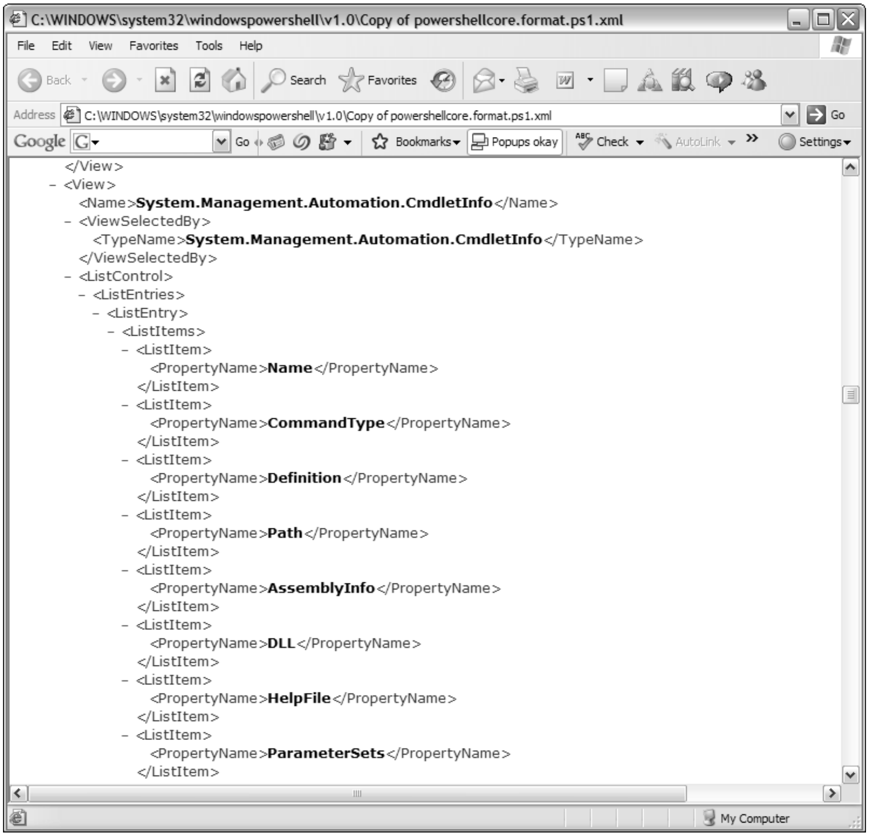


Figure 7-18

If you produce `CmdletInfo` objects for display, for example, by executing this command:

```
get-command get-childitem |
format-list
```

you can see in Figure 7-19 that the labels in each row of the output correspond to the `PropertyItem` elements shown in Figure 7-18.

In some situations, you can view information as a list by default. For example, execute this command:

```
(get-command get-childitem).Parametersets |
more
```

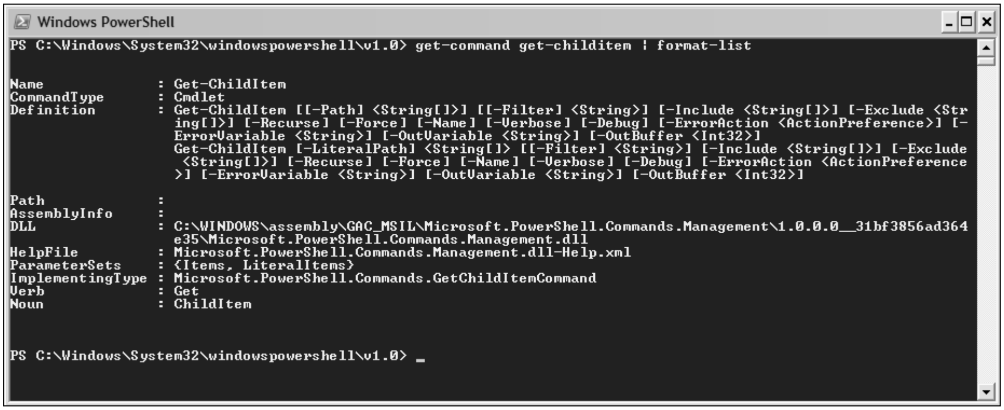


Figure 7-19

Information about the parameters in the parameter set(s) of the cmdlet of interest is displayed as a list, as you can see in Figure 7-20.

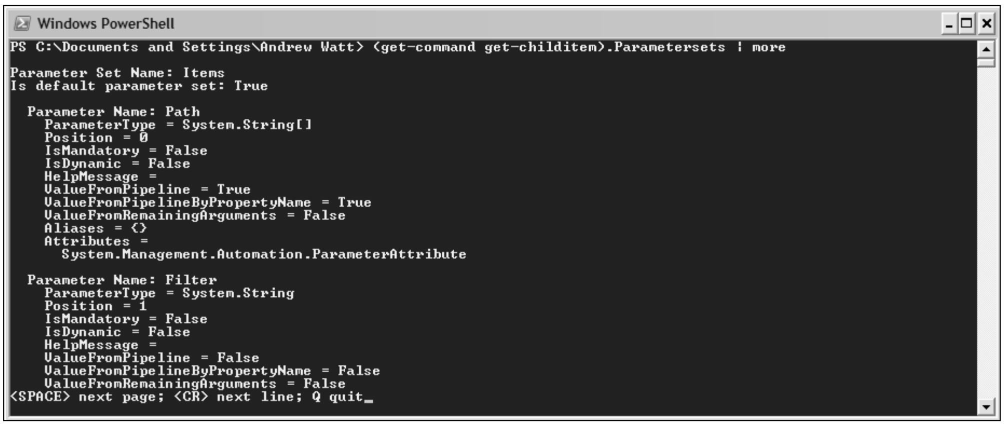


Figure 7-20

You can get detailed information about the behavior of the default formatter by exploring the content of the XML elements in `powershellcore.format.ps1xml`. More informally, to explore the behavior of the default formatter use this command:

```
get-command get-*
```

to find all the cmdlets which use a `get` verb. Then use each command piped to `more` to see the column headings provided by default.

Another approach to taking more control of the output from a pipeline is to use the `format-table` and `format-list` cmdlets that I describe in the following sections.

Using the format-table Cmdlet

The `format-table` cmdlet allows you to display information from a pipeline in a table. In some situations, the visual appearance produced by the `format-table` cmdlet is the same as that produced by the default formatter. If you use the `format-table` cmdlet with no properties specified for display the display is the same as the default output. You can confirm this by comparing:

```
get-process sql*
```

and:

```
get-process sql* |
format-table
```

As you can see in Figure 7-21, the displayed columns are the same.

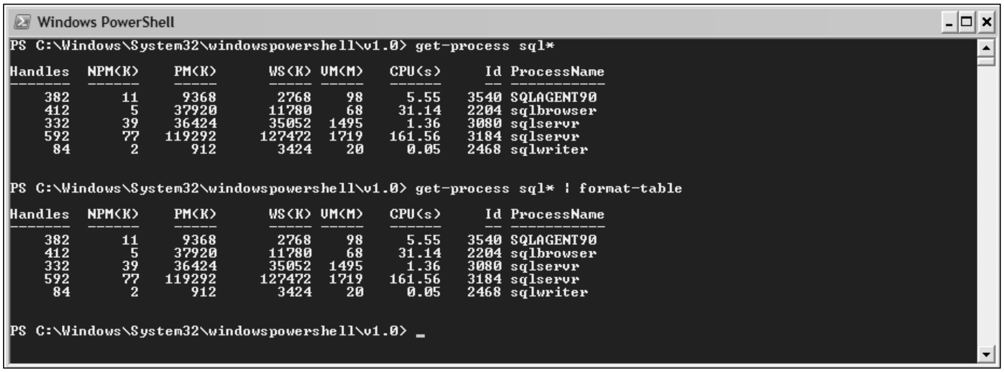


Figure 7-21

The usefulness of `format-table` is that it allows you to take control of the visual output to produce an appearance that is more selective and/or better laid out than the display produced by the default formatter. You do this by using parameters to modify the behavior of the `format-table` cmdlet.

The `format-table` cmdlet has several parameters, shown in the following list. Only the `property` parameter is a positional parameter.

- `property` — Specifies a property or list of properties to be displayed. You cannot use the `property` parameter if you use the `view` parameter in the same command.
- `AutoSize` — Specifies that the width of a column is to be adjusted automatically according to the width of the data.
- `HideTableHeaders` — If present specifies that the column headers are to be omitted.
- `GroupBy` — Specifies that output is to be grouped based on some shared property or value.
- `Wrap` — If present specifies that output is to wrap onto the next line. This contrasts with the default behavior, which is to truncate the content of a column if it exceeds the column width.

- `View` — Specifies the name of an alternate column view.
- `Force` — Overrides restrictions that will prevent the command succeeding.
- `InputObject` — Specifies an input object to be formatted. This is used when output is not being passed to the `format-table` cmdlet from an earlier pipeline step.
- `Expand` — Allows both information about a collection and its contained objects to be displayed.
- `DisplayErrors` — Specifies that errors are to be displayed on the command line.
- `showErrors` — Specifies that errors are to be passed along the pipeline.

In the following sections, I demonstrate how you can use several of these parameters.

Using the property Parameter

The `property` parameter is a positional parameter in position 1. The value of the property parameter is the name or, more usually, a comma-separated list of names, of properties of objects supplied from the pipeline.

This example uses the property parameter to selectively display information about the name, process ID, and handle count of processes whose name begins with `svc`. Type this code:

```
get-process -name svc*  |
format-table -property processname, ID, handlecount
```

or:

```
get-process svc* |
format-table processname, ID, handlecount
```

Figure 7-22 shows the results. Notice that text content is aligned left in a column and numeric content is aligned right.

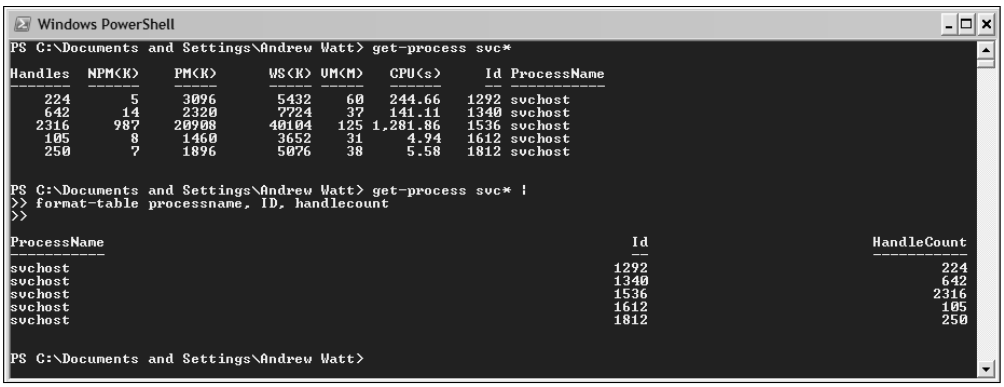


Figure 7-22

The first step of the pipeline retrieves all processes whose process name begins with the character sequence `svc`. The wildcard `*` matches zero or more characters that may follow.

The second step of the pipeline uses the `property` parameter positionally and is equivalent to:

```
format-table -property processname, ID, handlecount
```

By default, the `format-table` cmdlet spreads the columns corresponding to the supplied property names across the full width of the command window. When there are multiple rows, this can make reading the output difficult. One way to improve readability is to use the `autosize` parameter with `format-table`.

Using the autosize Parameter

The `autosize` parameter automatically adjusts the width of a displayed column to the greater of the width of the column label or the column content. Generally, this makes it easier to read along rows.

This example uses the autosize (or `-auto`, or even just `-a`) parameter to display the previously selected properties in a more compact display. The width of a column is adjusted to correspond to the width of the data it contains. Type:

```
get-process svc*|
format-table -property processname, ID, handlecount -autosize
```

Figure 7-23 shows the output. Notice that when you use the `-autosize` parameter, the three columns of data are displayed closer together, thus improving readability along a line of data.

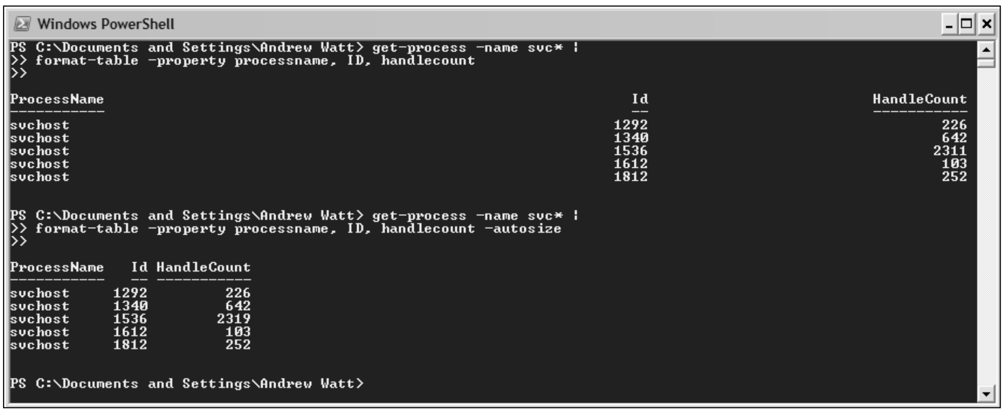


Figure 7-23

The only change from the preceding example is the presence of the `autosize` parameter. Its value is a boolean. You don't need to supply a value, but the Windows PowerShell parser allows you to supply a value using the colon notation, `-autosize:$true`, if you prefer. Simply providing the name of the `-autosize` parameter indicates that the value of the parameter is `$true`.

Hiding Table Headers

`Format-table`'s `-hidetableheaders` parameter allows you to hide the headers for each column that Windows PowerShell displays. To hide the column headers for the preceding example, simply add the `hidetableheaders` parameter:

```
get-process svc*|
format-table -property processname, ID, handlecount -autosize
-hidetableheaders
```

Figure 7-24 shows the result together with the result from the preceding example.

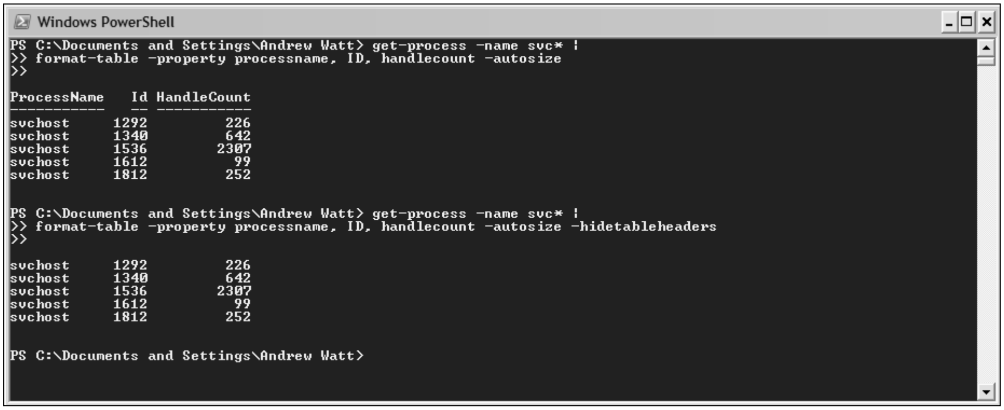


Figure 7-24

Grouping Output

The `groupby` parameter allows you to group output from the `format-table` cmdlet. Visually, the output looks like a series of small tables, some of which may have only a single line.

This example shows how to group output from the `get-process` cmdlet by using the `groupby` parameter of the `format-table` cmdlet. It retrieves information on all processes that begin with `sq` and groups the display by process name. Amend the command if you don't have SQL Server installed.

```
get-process sq*|
format-table -groupby processname
```

For some processes, for example, `sqlservr`, there are multiple processes retrieved, as shown in Figure 7-25.

The first step of the pipeline is familiar if you have read the examples earlier in this chapter. Because you supply the `groupby` parameter in the second step of the pipeline, the `format-table` cmdlet groups the information by process name before displaying it. Since no `property` parameter was used with `format-table`, the displayed columns are the default ones that `format-table` uses with the `get-process` cmdlet.

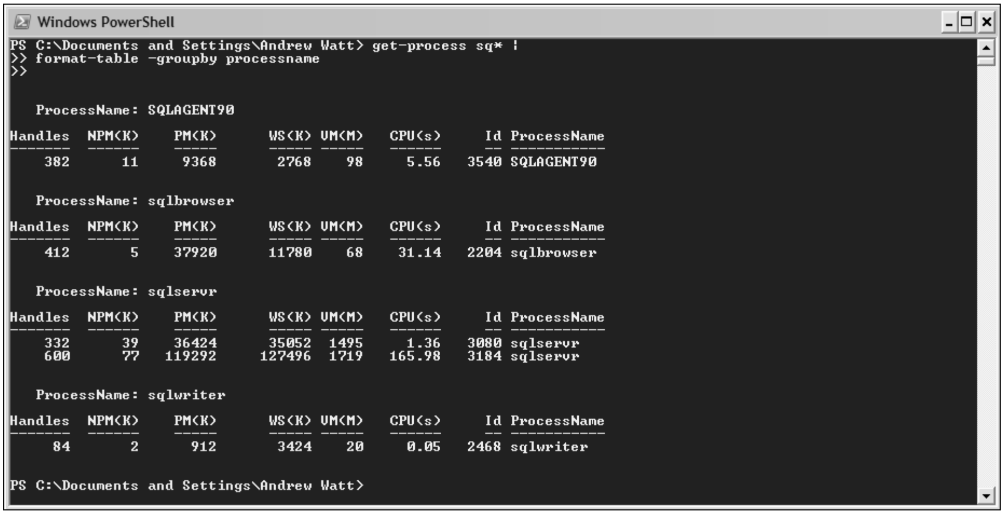


Figure 7-25

When you use the `-groupby` parameter with the `format-table` cmdlet, the output differs from that you see when you use the `group-object` cmdlet in a separate pipeline step:

```
get-process sq* |
group-object Name |
format-table
```

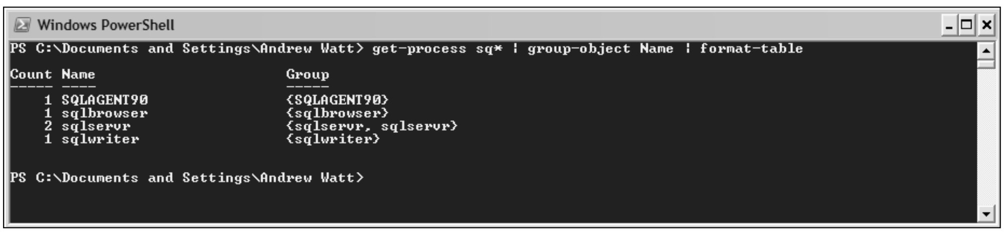


Figure 7-26

Figure 7-26 shows the result of executing the preceding command. By comparing it to Figure 7-25, you can see that the output displayed is significantly different. When you use the command shown in Figure 7-25, `System.Diagnostics.Process` objects are presented to the `format-table` cmdlet. In the most recent command, `Microsoft.PowerShell.Commands.GroupInfo` objects are presented to the `format-table` cmdlet.

Specifying Labels and Column Widths

The `format-table` cmdlet allows you to provide custom names for each column (if, for example, you find the corresponding property name isn't ideal for your users) and to specify the width of each column. Used well, this allows you to create a significantly improved visual display.

To do this, you specify an associative array, which contains a comma-separated list of values, to the property parameter. Each value specifies an expression that defines the content of a column, a column width, and a label to be used as the column header and takes this form:

```
@{expression = "anExpression"; width = aNumber; label = "aString"}
```

This approach is most useful when you run a script repeatedly and want to display the output in an easily read format.

This example shows you how to display the process name, ID, and handle count of processes whose name begins with `svc`, while specifying custom labels for the columns and specifying the width of each column. The label for the first and third columns simply splits the property name into two words.

Type the following command:

```
get-process svc* |
format-table @{expression="processname"; width=15; label="Process Name"},
@{expression="ID"; width=10; label = "ID"},
@{expression = "handlecount"; width=15; label = "Handle Count"}
```

Figure 7-27 shows the results. Depending on the data, this can give you a much more readable output than, for example, using the `autosize` parameter. The column header is customized.

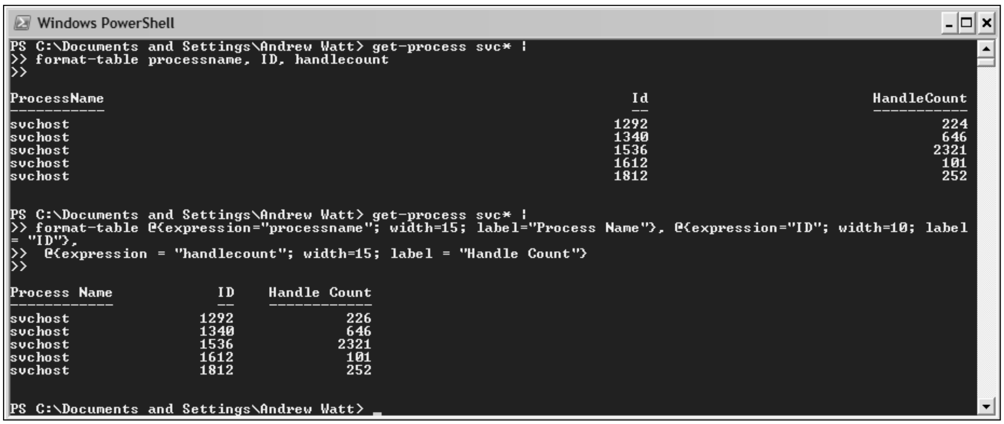


Figure 7-27

The first step of the pipeline retrieves processes whose `processname` begins with the character sequence `svc`.

The second step provides three values in a hash table whose values are a comma-separated list for the `property` parameter of the `format-table` cmdlet. The first value:

```
@{expression="processname"; width=15; label="Process Name"}
```

has three parts. The `expression` part specifies which property (or expression) is to supply data for the column. The `width` part specifies the width for that column. The `label` part specifies the column name to be displayed.

Be careful not to enclose the value of `width` in paired quotes or an error message telling you that the type of the value is wrong will be displayed. The value of the column part must be an integer, not a string.

Using the format-list Cmdlet

The `format-table` cmdlet is useful when the values to be displayed are short or values to be displayed are few. But when there are many values to be displayed or if individual values are long then using the `format-table` cmdlet can produce unsatisfactory output. For example, if you wanted to see all the properties returned by the `get-childitem` cmdlet in a table you would use a command like this:

```
get-childitem |
format-table *
```

Figure 7-28 show the type of results you will see. As you can see, the values in many columns are truncated to the point of being useless as a source of information to the user.

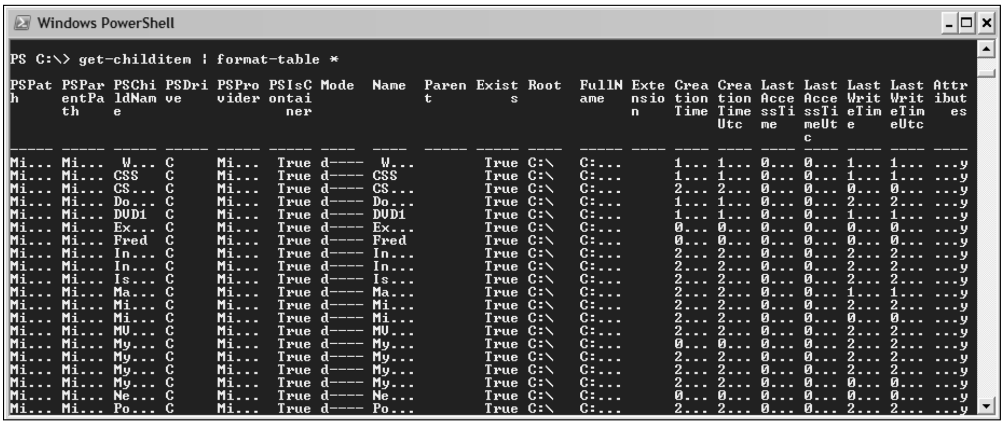


Figure 7-28

In this situation, Windows PowerShell, by default, displays too many columns onscreen. The `format-list` cmdlet allows you to better display all the information onscreen.

This example uses the `format-list` cmdlet to ensure that the complete value of each property is displayed for each process.

Type this command:

```
get-childitem |
format-list *
```

The first part of the output is shown in Figure 7-29. You can now see the value for each property without any truncation. The downside is that the information is spread across many screens of information — but at least you can view the information you want.

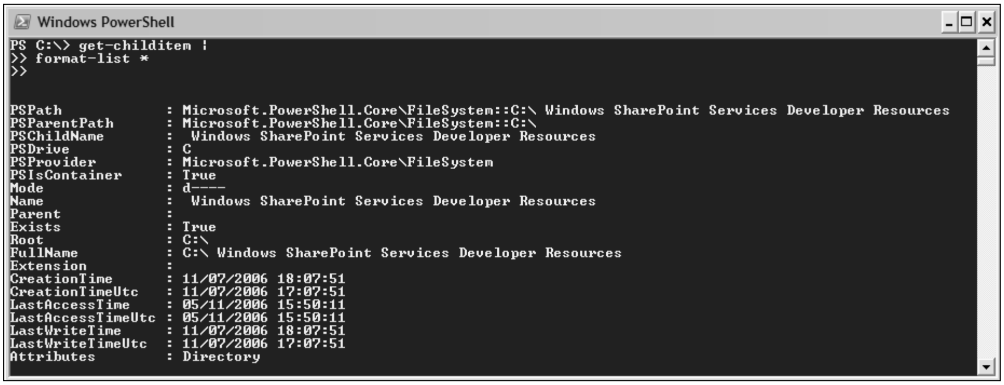


Figure 7-29

Using the update-formatdata and update-typedata Cmdlets

PowerShell version 1.0 format files have the file extension `.ps1xml`. I described earlier in this chapter a little of the structure of a format file and mentioned that the formatting information used by the default formatter is contained in the `powershellcore.format.ps1xml` file. The file contains many XML elements but also contains a digital signature. You may want to have other format files available for use. The `update-formatdata` cmdlet is intended to allow you to load other `ps1xml` files into the Windows PowerShell shell.

The `update-formatdata` cmdlet supports the following parameters, in addition to the common parameters:

- `appendPath` — Specifies a path to optional format files that are processed after the built-in format files are loaded
- `prependPath` — Specifies a path to optional format files that are processed before the built-in format files are loaded

The `update-typedata` cmdlet is similar in concept to the `update-formatdata` cmdlet. Formatting data for types is held in the `types.ps1xml` file. The `update-typedata` cmdlet allows you to load additional files containing format data for the display of types.

The `update-typedata` cmdlet supports the following parameters, in addition to the common parameters:

- `appendPath` — Specifies a path to optional `type.ps1xml` files that are processed after the built-in files are loaded
- `prependPath` — Specifies a path to optional `type.ps1xml` files that are processed before the built-in files are loaded

Summary

The `where-object` cmdlet allows you to filter objects passing along a pipeline to reduce or eliminate unwanted results. The value of the `-filterScript` parameter is used to determine whether an object is passed along the pipeline or is discarded.

The `select-object` cmdlet allows you to select specified objects or properties for further processing in a pipeline. The `-first` and `-last` parameters allow you selectively to process a specified number of elements at the beginning or end of an array. When used with sorted data these parameters allow you to select a specified number of the highest or lowest values in a data set.

The formatting of objects for display in Windows PowerShell is carried out using the default formatter. The `format-table` cmdlet allows you to more selectively display data or to customize the appearance of selected data. The `format-list` cmdlet allows you to display data in a list format.

8

Using Trusting Operations

Tools such as Windows PowerShell provide tremendous power. But at the same time, one potentially terrifying thing about Windows PowerShell is that its power makes it potentially more destructive if you do something wrong. Imagine that you want to delete some files or stop some processes or services depending on the value returned by an expression. You really need to be sure of what you are doing, don't you? You don't want to end up deleting some crucial files on which your company depends just because you made a mistake in the syntax on the command line or in a Windows PowerShell script.

The Windows PowerShell designers have that base covered by providing several options to use with cmdlets that let you check the effects of what you plan to do. I describe these options in this chapter.

There are three parameters available for use with many, but not all, Windows PowerShell cmdlets that allow you to anticipate exactly what a command will do or monitor what a command has done. The cmdlets that lack these parameters cannot change system state. The parameters are:

- `whatif` — Allows you to see what a command would have done without actually executing the command
- `confirm` — Allows you to see the individual actions a command would have taken and allows you to confirm or cancel each action
- `verbose` — Allows you to see in detail what you have done

Some of the examples in this chapter are potentially damaging to your system. Please be VERY CAREFUL when you type the code examples to ensure that you do not unintentionally run potentially damaging code. And when you extend or adapt the examples, make liberal use of the `-whatif` parameter to check that your adaptations don't have unintended effects.

In addition, while you are learning the effects of Windows PowerShell commands, you may want to focus your experimentation on a test machine.

Look Before You Leap

The `whatif`, `confirm`, and `verbose` parameters are available on many, but not all, Windows PowerShell cmdlets. Strictly speaking only the `whatif` and `confirm` parameters give you the information you would like *before* you leap. The `verbose` parameter tells you that you have leapt and exactly what you hit on the way down! Sometimes that after-the-event information you get from the `-verbose` parameter will be all you need. If that isn't enough, then you probably need to use the `whatif` or `confirm` parameters.

The cmdlets that are potentially most dangerous are those that use the `remove` verb. There are five such cmdlets:

- `remove-drive`
- `remove-item`
- `remove-pssnapin`
- `remove-property`
- `remove-variable`

In later sections in this chapter, I demonstrate how you can use the `-whatif` and `-confirm` parameters with some of the preceding cmdlets.

Using the remove-item Cmdlet

The `remove-item` cmdlet deletes an item from a provider. Two important uses are the deletion of items (folders and files) in the file system and the deletion of items in the registry. At the risk of stating the obvious, deletions in the file system or registry can produce undesired effects.

In addition to the common parameters (covered in Chapter 6), the `remove-item` cmdlet supports the use of the following parameters:

- `path` — Specifies the path of the item(s) to be removed. A positional parameter in position 1.
- `recurse` — If present, specifies recursive interpretation of the command. Descendant items, not only child items, of the current location are removed.
- `force` — If present, overrides restrictions such as file renaming.

- `include` — If present, specifies items to include. The value of this parameter qualifies the value of the `-path` parameter.
- `exclude` — If present, specifies items to exclude. The value of this parameter qualifies the value of the `-path` parameter.
- `filter` — Specifies a filter that qualifies the value of the `-path` parameter.
- `credential` — If present, specifies a credential to use to gain access to the item(s).

> **In the examples that follow, you perform some destructive actions (deleting files, etc.). Therefore, I strongly suggest that you either work on test directory structures until you are sure what you are doing or use the `whatif` parameter (described later in this chapter).**

Since the `remove-item` cmdlet can be destructive, you should first create a folder and file structure that you can safely use `remove-item` on. If you prefer, you can create a similar structure by using Windows Explorer and Notepad. Since Windows PowerShell does all you need, it makes sense to me to use Windows PowerShell cmdlets to get the task done.

In this example, I will show you how to use Windows PowerShell to create a folder and file structure that you can then use the `remove-item` cmdlet on in later examples in this chapter.

In the examples that follow, I am using the C: drive to hold the test files. Feel free to change the drive or folder names to suite your setup.

Start Windows PowerShell, and type the following to create a new directory named `Disposable`:

```
new-item -path c:\Disposable -type directory
```

The value of the `-path` parameter specifies the location of the new item. The value of the `-type` parameter specifies that you are creating a folder (aka a directory). Figure 8-1 shows the result. Notice that a new folder named Disposable has been created.

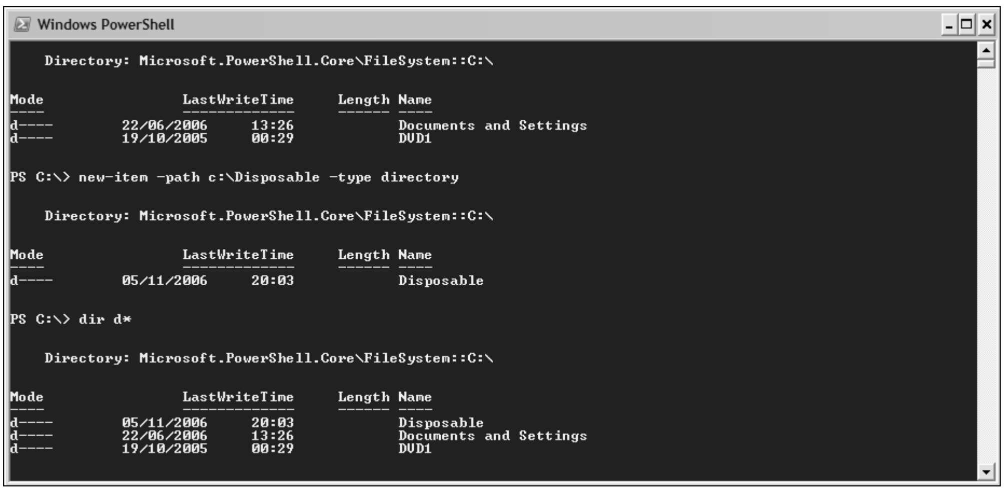


Figure 8-1

Next add some text files that will be used to test the use of the `include` and `exclude` parameters with `remove-item` in later examples.

To create four simple test text files in the `Disposable` directory, type these commands. The commands assume that C:\Disposable is the current working directory.

```
"This is test 1" > c:\disposable\Test1.txt
"This is test 2" > c:\disposable\Test2.txt
"This is test 3" > c:\disposable\Test3.txt
"This is test 4" > c:\disposable\Test4.txt
```

Confirm that you have created the desired files by using this command:

```
get-childitem c:\Disposable\*.txt
```

The result showing the successful creation of four sample files should look like Figure 8-2.

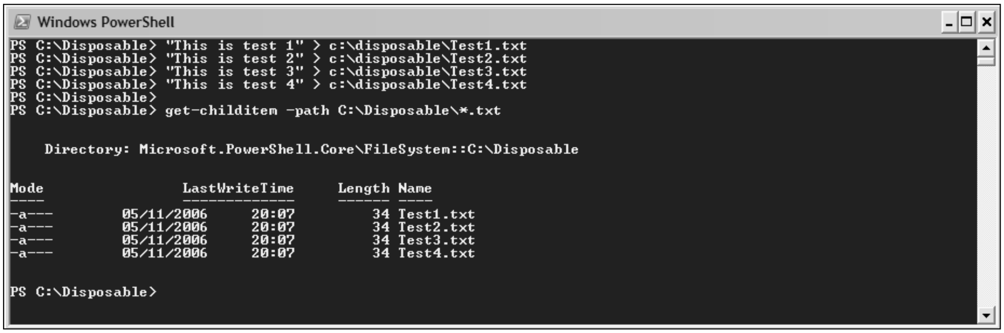


Figure 8-2

To help demonstrate the use of the `-recurse` parameter later in the chapter, you should also create an additional folder named `subfolder` inside the `Disposable` folder, using the following command:

```
new-item -path c:\Disposable\subfolder -type directory
```

Figure 8-3 shows that the subfolder folder has been successfully created.

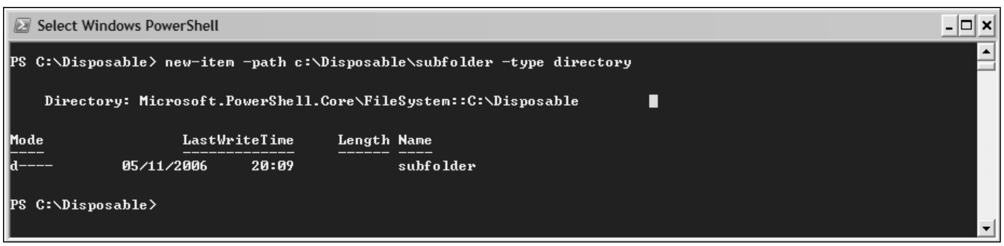


Figure 8-3

Switch to the subfolder directory, and then add some simple text files, using the following commands:

```
"This is test 1" > c:\disposable\subfolder\Test1.txt
"This is test 2" > c:\disposable\subfolder\Test2.txt
"This is test 3" > c:\disposable\subfolder\Test3.txt
"This is test 1 backup." > c:\disposable\subfolder\Test1.bak
"This is test 2 backup." > c:\disposable\subfolder\Test2.bak
"This is test 3 backup." > c:\disposable\subfolder\Test3.bak
```

To confirm that you have created the desired six files in the `c:\Disposable\subfolder` folder, use this command:

```
get-childitem c:\Disposable\subfolder
```

Figure 8-4 shows the desired result.

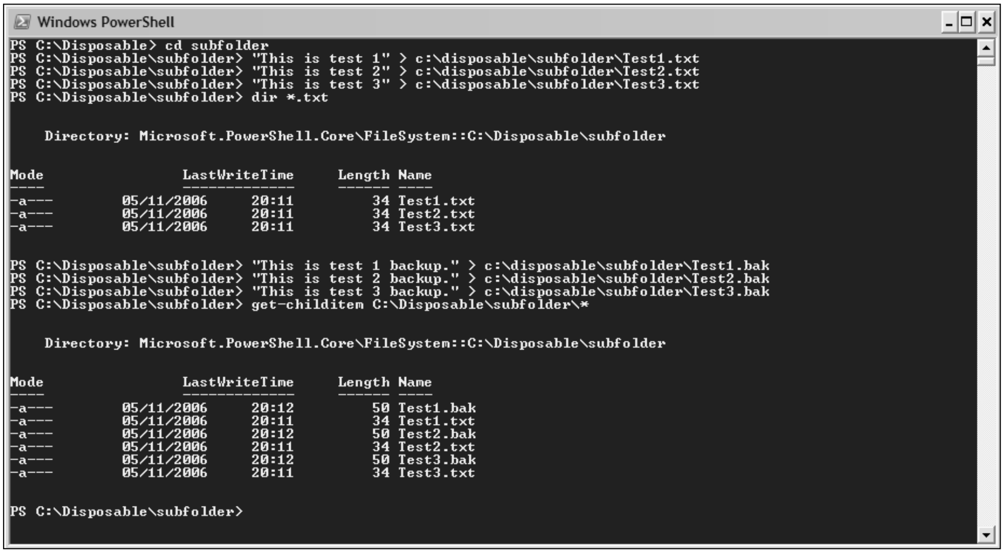


Figure 8-4

Finally, copy the `Disposable` folder and its files and nested folder, so you can recreate the structure by copying back the copy. Use the following command:

```
copy-item -path c:\Disposable -destination c:\DisposableCopy -recurse
```

Now that you have saved a copy of the directory structure, you can use the `remove-item` cmdlet to remove items. First, let's look at what the preceding commands have done. The command

```
new-item -path c:\Disposable -type directory
```

uses the `new-item` cmdlet to create a new folder. The value of the `-type` parameter specifies that it is a folder that is to be created. The value of the `-path` parameter specifies what folder is to be created.

Redirection Operators

In Windows PowerShell, you can redirect content to a file using redirection operators. To redirect so that a new file is created or an existing file is overwritten, use the > operator. To redirect so that content is appended to an existing file (if it exists) or a new file is created (if the file doesn't already exist), use the >> operator.

The command

```
"This is test 1" > c:\disposable\Test1.txt
```

takes a literal string and redirects the output from the console (the default) to a file named `c:\DisposableTest1.txt`. The > character is the redirection operator in Windows PowerShell. The similar commands create the other three test files in the `Disposable` directory.

The command

```
new-item -path c:\Disposable\subfolder -type directory
```

is similar to the command that created the `Disposable` directory. The value of the `-type` parameter specifies that a folder is to be created. The value of the `-path` parameter specifies the location of the new folder.

The test files in the `subfolder` folder are created in the same way as the test files in the Disposable directory.

The command

```
copy-item -path c:\Disposable -destination c:\DisposableCopy -recurse
```

uses the `copy-item` cmdlet to create a complete copy of the Disposable folder and all its content in a folder named `c:\DisposableCopy`. The value of the `-path` parameter specifies what is to be copied. The value of the `-destination` parameter specifies where the copy is to be located. The presence of the `-recurse` parameter specifies that the copying is to be done recursively.

If you omit the `-recurse` parameter and simply type

```
copy-item -path c:\Disposable -destination c:\DisposableCopy
```

then a folder named `DisposableCopy` is created, but it is empty.

Now that the test folder and file structure has been created, you can try using the `remove-item` cmdlet. In this example, you delete the file named `test3.bak` in the `subfolder` directory.

To delete a single file, you use the `remove-item` cmdlet and specify the file to be deleted as the value of the `-path` parameter. Since the `-path` parameter is a positional parameter, you can omit the name of the parameter if you want to. The command is shown here with the `whatif` parameter for safety:

```
remove-item c:\Disposable\subfolder\test3.bak -whatif
```

Run the command with the `-whatif` parameter. Notice the message displayed in Figure 8-5. Then run the command again without the parameter:

```
remove-item c:\Disposable\subfolder\test3.bak
```

to actually delete the file `Test3.bak`.

To confirm that `test3.bak` has been deleted, use this command:

```
get-childitem c:\Disposable\subfolder\*.bak
```

Figure 8-5 shows the result before and after the deletion of a single file.

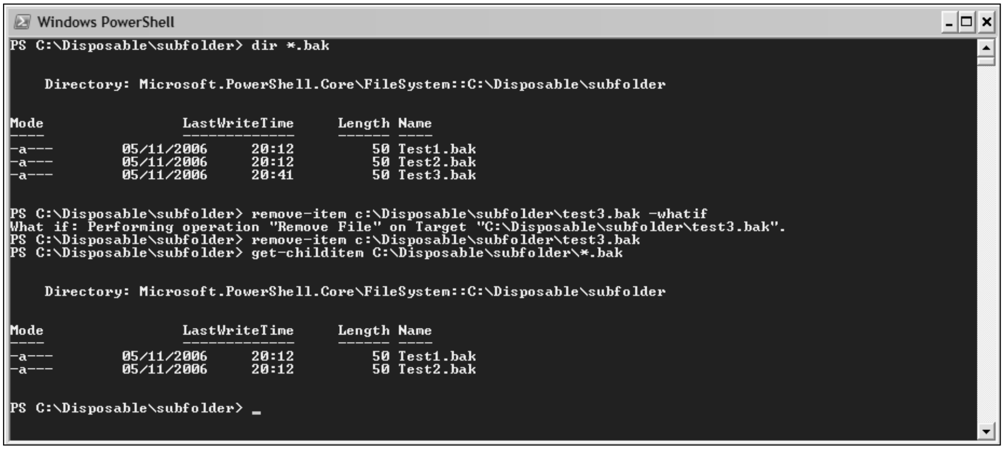


Figure 8-5

The command

```
remove-item c:\Disposable\subfolder\test3.bak -whatif
```

does not actually delete anything — rather, it tells you what files would have been deleted if the `-whatif` parameter had not been specified. In this case, only a single file, `Test3.bak`, would have been deleted. Removing the `-whatif` parameter like this:

```
remove-item c:\Disposable\subfolder\test3.bak
```

deletes the file. Notice in Figure 8-5 the difference between the files listed before and after the preceding command is run.

You can also use wildcards to get Windows PowerShell to delete multiple files at one time. The single `*` wildcard, for example, matches zero or more characters (that is all files — so use this wildcard with care). The `?` wildcard matches a single character.

In the next example, you use a wildcard to delete two files in the subfolder directory: `test2.txt` and `test2.bak`.

First, confirm that the relevant two files exist, using the following command:

```
get-childitem c:\Disposable\subfolder\test2.*
```

Type the following command:

```
remove-item -path c:\Disposable\subfolder\test2.* -whatif
```

to test what the command does. Notice in Figure 8-6 that a message is displayed for each of the two files that the command would delete if the `-whatif` parameter were not present. Next, type

```
remove-item -path c:\Disposable\subfolder\test2.*
```

to delete the desired two files.

Then type

```
get-childitem c:\Disposable\subfolder\test2.*
```

to confirm that the files have been deleted. Now no files match the pattern test2.*. In other words, the two files have been successfully deleted. As you can see in Figure 8-6, the previously listed files have been deleted.

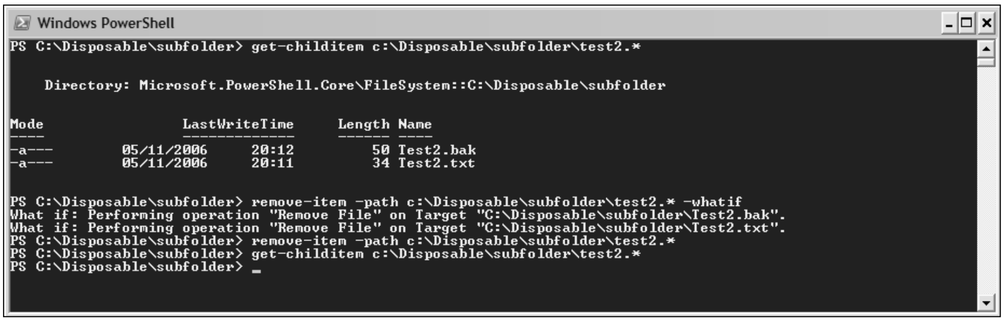


Figure 8-6

Delete the `Disposable` folder by using Windows Explorer or the following Windows PowerShell command:

```
remove-item C:\Disposable
```

Copy the `DisposableCopy` folder to the `Disposable` folder, using this command:

```
copy-item C:\DisposableCopy C:\Disposable -recurse
```

to recreate the `Disposable` folder and its contents. Move to the `C:\Disposable\subfolder` folder.

The key command in this example is:

```
remove-item -path c:\Disposable\subfolder\test2.*
```

The value of the `path` parameter includes the `*` wildcard. The `*` wildcard matches any characters, so it matches the files ending `bak` and `txt`. Therefore both `Test2.bak` and `Test2.txt` are deleted by the `remove-item` cmdlet.

The `-include` parameter allows you to tighten up a choice specified in the `-path` parameter of the `remove-item` cmdlet.

Execute the following command:

```
remove-item -path * -whatif
```

Notice in Figure 8-7 that all six files in the folder would be deleted. Add an `-include` parameter, as in the following command:

```
remove-item -path * -include *.txt -whatif
```

and execute it. Notice in Figure 8-7 that only the files that match the `*.txt` in the value of the `-include` parameter would be deleted.

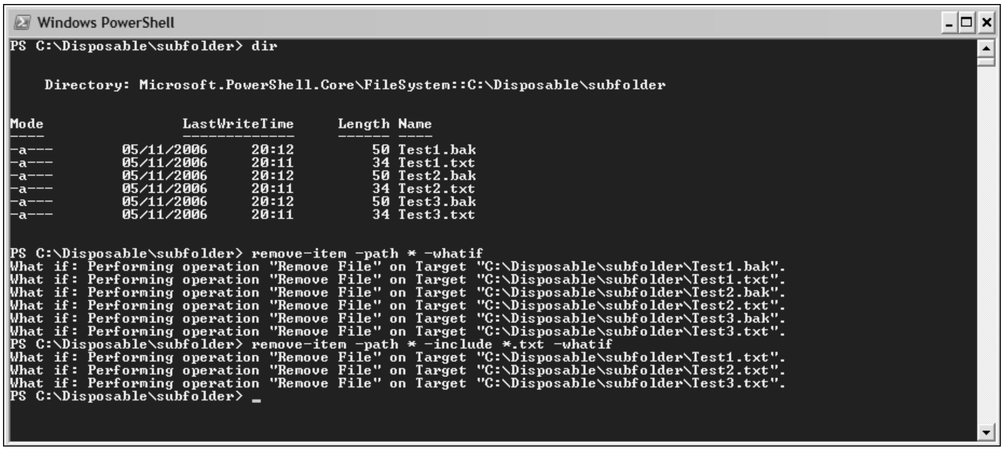


Figure 8-7

When you come to use the `-recurse` parameter, you need to be aware of semantics that not all early users of Windows PowerShell find intuitive. Requests were made during the beta program that the semantics be changed. However, in the final release of Windows PowerShell version 1.0 the semantics seem to me to be unexpected in some situations.

To demonstrate an example of the issue that you need to be aware of, change to the `C:\Disposable` directory. Execute this command:

```
get-childitem -path * -include *.txt
```

Notice in Figure 8-8 that `.txt` files in the Disposable folder are selected.

Similar behavior is seen if you execute the following command:

```
remove-item path * -include *.txt -whatif
```

With these values for their respective `-path` and `-include` parameters, the `get-childitem` and `remove-item` cmdlets appear to behave consistently with each other.

However, if you add the `-recurse` parameter to both the preceding commands:

```
get-childitem -path * -include *.txt -recurse
```

and:

```
remove-item -path * -include *.txt -whatif -recurse
```

the behavior diverges significantly, as you can see in Figure 8-9. The `-recurse` of the `get-childitem` cmdlet behaves as I would expect it to. The `-recurse` parameter of the `remove-item` cmdlet doesn't behave as I would expect.

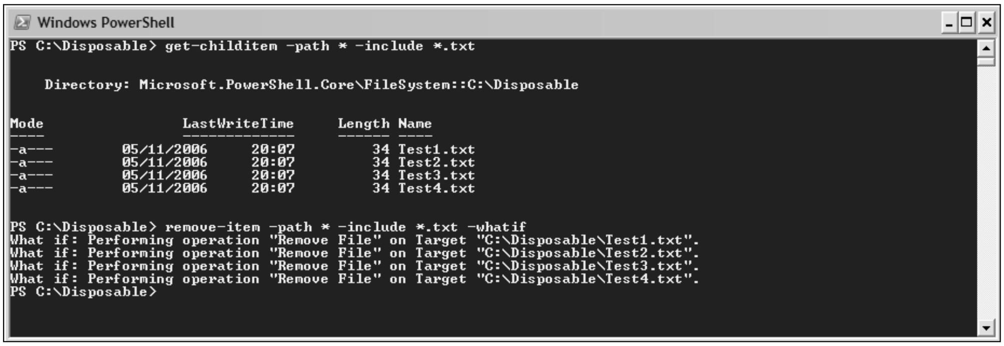


Figure 8-8

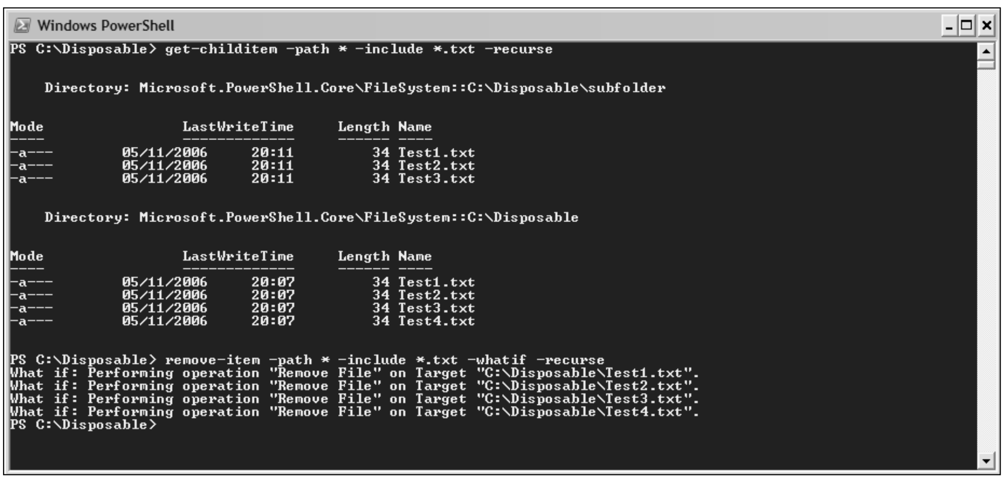


Figure 8-9

If you intend to use the `remove-item` cmdlet in the registry be very sure that you know what you are doing.

Using the whatif Parameter

I think that the `whatif` parameter is an incredibly valuable part of the Windows PowerShell approach. It allows you to test the effect of a command before anything is changed on a machine. This is very useful if, as shown in earlier examples, you use the `remove-item` cmdlet with a wildcard.

Another situation where the `whatif` parameter is useful is with the `stop-process` and `stop-service` cmdlets.

Using the stop-process Cmdlet

You can use the `stop-process` cmdlet to stop one or more running processes on a machine. But for obvious reasons, this can be dangerous, and Windows PowerShell builds in a couple of safety mechanisms. First, is the `-whatif` parameter I describe below. But the `-whatif` parameter isn't the only safety mechanism built into the `stop-process` cmdlet. Imagine that you type a command like the following:

```
stop-process
```

If you were familiar with the behavior of the `get-process` Cmdlet, you might expect the command to accept a default to stop all running processes — since this would not be a very safe approach, fortunately this is not what happens. In this case, `stop-process` prompts you for a process ID for a process to stop, as shown in Figure 8-10.

Figure 8-10

The `stop-process` cmdlet has `-ID`, `-processname`, `-input`, and `-passthru` parameters. The `-ID` parameter is the only positional parameter and is recognized in position 1. So, when you type

```
stop-process
```

the Windows PowerShell interpreter needs a positional parameter to be supplied, in this case the process ID property of one or more running processes.

The following examples use the `-whatif` parameter. Omitting it may cause your machine to crash, depending how you adapt the examples.

In this example, I start a Notepad process from the command line, find its ID, and then stop it using the value of its ID property.

To start Notepad from the Windows PowerShell command line, type this command at the Windows PowerShell console:

```
notepad
```

A Notepad window opens.

To find the ID value of all running instances of Notepad, type:

```
get-process -processname notepad
```

This command displays some limited information to the console about all the running instances of Notepad on your system.

To stop a particular Notepad instance, take note of the value of its ID property. On my machine when I wrote this, the value of Notepad's process ID was `2692`, so I would type:

```
stop-process -ID 2692
```

To confirm that the Notepad process has exited, type:

```
get-process Notepad
```

Figure 8-11 shows the results you see onscreen after each command is typed. I have several Notepad processes running, but ID 2692 is no longer among them, since it has been stopped.

Typing

```
notepad
```

simply launches a new instance of the Notepad application. You don't need to use the `new-object` cmdlet to start Notepad. However, the `new-object` cmdlet can be used to launch other COM applications. See Chapter 13 for further information.

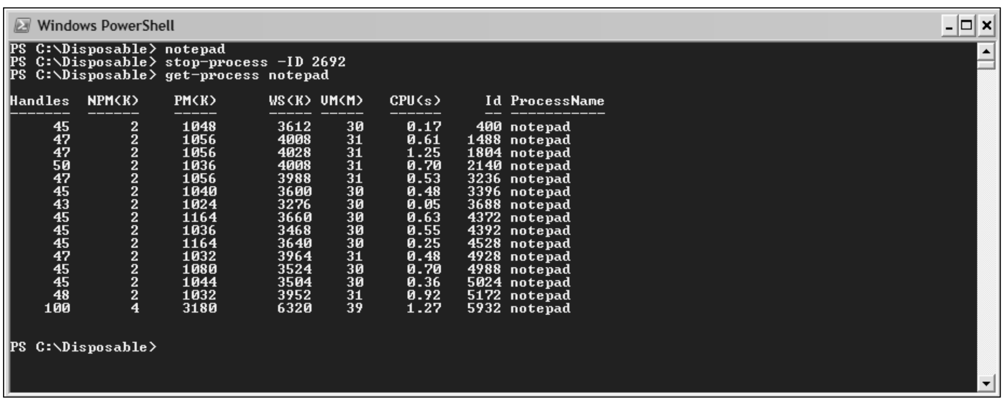


Figure 8-11

The command

```
get-process -processname notepad
```

is used to retrieve information about all the running Notepad processes. The value of the `-processname` parameter specifies that only objects whose `processname` property is equal to `Notepad` are returned. The result displays the value of the `Id` property for any running Notepad process.

Supplying a valid value of a running Notepad process as follows (on my machine):

```
Stop-Process -ID 5268
```

results in the specified process being stopped.

If you want to stop multiple processes, one way to do it is to simply provide a comma-separated list of `Id` values, assuming that you know the relevant ID values. Alternatively, you can use wildcards to limit the selection of processes to stop and use the `whatif` parameter to refine the command until it does just what you want it to.

This example shows you how you can use Windows PowerShell to stop multiple processes using the `whatif` parameter. The example intends to stop all processes relating to Microsoft SQL Server. If you don't have convenient access to SQL Server, you could start a number of Notepad instances and adapt the command to fit that situation.

Without using the `-whatif` parameter, you could use this command to find all SQL Server processes:

```
get-process *sql*
```

Then you could inspect the results, which might look like Figure 8-12, to see if the desired processes are selected.

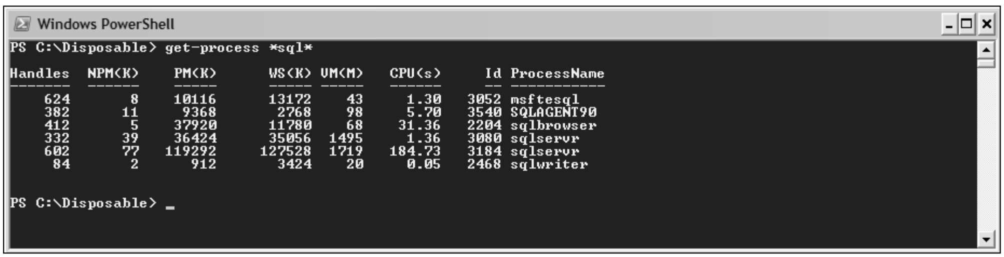


Figure 8-12

If, after inspecting the results of the `get-process` cmdlet, you think you have a wildcard that does exactly what you want, you can simply use the same wildcard combination with the `stop-process` cmdlet:

```
stop-process -processname *sql*
```

The `–name` *parameter of the* `get-process` *cmdlet is positional, so you don't need to supply the name of the parameter. In the* `stop-process` *cmdlet, the* `–name` *parameter is a named parameter (the* `–ID` *parameter is its only positional parameter), so when specifying process names you need to provide the parameter's name, too.*

The `-whatif` parameter provides an alternative and, in my opinion, a better approach. Type

```
stop-process -name *sql* -whatif
```

and you can see, as shown in Figure 8-13, the processes that will be stopped.

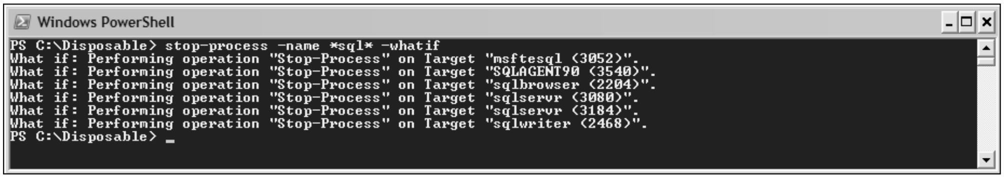


Figure 8-13

Using the `-whatif` parameter only displays the processes that the `stop-process` cmdlet would stop; no processes are stopped when the `-whatif` parameter is present.

To stop the chosen processes, simply repeat the command, deleting the `-whatif` parameter. When you run the command without the `-whatif` parameter, the processes are stopped.

Using the stop-service Cmdlet

The `stop-service` cmdlet is similar to `stop-process`; however, it only stops running services as opposed to all services. You can use the `-whatif` parameter with the `stop-service` cmdlet to make sure that you don't stop any services that you intended to allow to continue.

In addition to the ubiquitous parameters, the following parameters are available for use with the `stop-service` cmdlet:

- `name` — Specifies the name of the service. Cannot be used with the `displayname` parameter in the same command.
- `include` — Specifies which items the cmdlet will act on.
- `exclude` — Specifies which items the cmdlet will not act on.
- `force` — Allows the cmdlet to override dependency restrictions.
- `passthru` — Takes the object created as a result of the cmdlet and passes it down the pipeline
- `displayname` — Specifies the display name of the service. Cannot be used with the `name` parameter in the same command.

If you are unclear about the differences between the name of a service, as specified by the `-name` parameter, and the display name, as specified by the `-displayname` parameter, execute the following command to see the difference.

```
get-service * |
format-list Name, DisplayName
```

This example allows you to stop any services relating to SQL Server. If you don't have SQL Server installed, adapt the name of the services to be displayed. It also illustrates one important limitation of the `-whatif` parameter. While the `-whatif` parameter tells you what operation it would attempt, there is no indication of whether the operation would be successful. You might not, for example have the rights to stop a service. Thus, `-whatif` may tell you that it would stop a service, but in fact when you try, the command fails.

First, find all SQL Server services on the machine and sort them according to their status (running or stopped), using this command:

```
get-service *sql* |
sort-object status
```

Figure 8-14 shows the results on a machine with SQL Server 2005 components installed. Notice that 7 of 8 services are running.

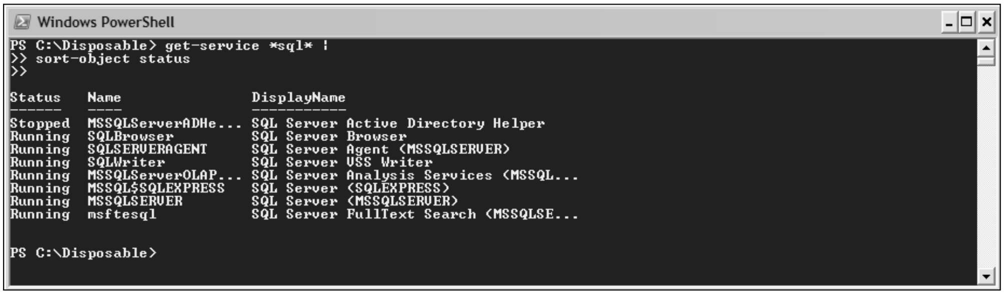


Figure 8-14

You can use the `-whatif` parameter with the stop-service cmdlet in this command:

```
stop-service -servicename *sql* -Whatif
```

In Figure 8-15, you can see that it indicates that it will stop eight processes, although one of those services is already stopped.

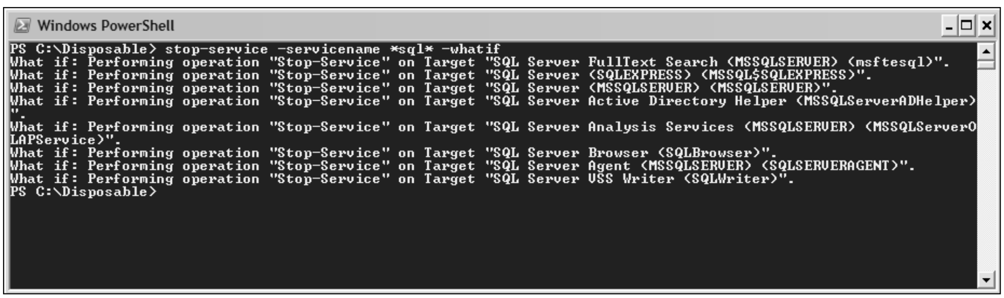


Figure 8-15

The wildcards in the value of the `servicename` parameter of the `stop-service` cmdlet will match any SQL Server service. The `-whatif` parameter does not check if the service is actually running before indicating that the `stop-service` cmdlet will stop it. Practically, this may not be too much of a problem, since you quite possibly want all services stopped anyway. The one practical issue that can arise is that the additional services displayed (which are already stopped) can make it more difficult to read the effect of the `stop-service` cmdlet.

Using the confirm Parameter

The `-confirm` parameter allows you to step through the processing of a cmdlet and decide at each point whether to allow it to implement the intended action or to prevent it taking that action.

When you use the `confirm` parameter, you are offered the options of `Yes (Y)`, `Yes to All (A)`, `No (N)`, `No to all (L)`, `Suspend (S)`, and `Help (?)`.

This example repeats the preceding example but uses the `Confirm` parameter in place of the `whatif` parameter to demonstrate the difference in behavior. Notice that with the `-confirm` parameter you never get an overview of what you are going to do. You need to evaluate each action individually. However, when using the `-whatif` parameter, if you find that you can see the actions you want, then changing to the `-confirm` parameter allows you to step through the available actions, confirming those that you want to take place and rejecting those that you don't want.

Type the following command:

```
stop-service -servicename *sql* -confirm
```

Figure 8-16 shows the result after responding `No` to the first option offered and before making a decision about the second option.

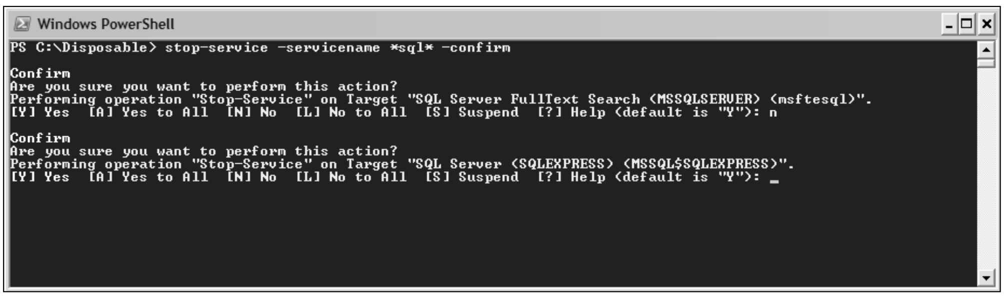


Figure 8-16

The first part of the statement:

```
stop-service -servicename *sql*
```

would stop several SQL Server services if the `-confirm` parameter weren't present. For each service, you are asked to specify whether you want to stop that service, not stop that service, stop all services, or decline stopping any service. Be careful with the `Yes to All` option: you must be certain which services will be affected before using it.

The Suspend option offered when you use the `-confirm` parameter allows you to get a subshell. Activity in the current shell is suspended. You can issue other commands in the subshell that might help you decide what you want to do, then type Exit (to exit the subshell) to return to the original command and decide what you want to do.

Using the verbose Parameter

The `-verbose` parameter tells you what has been done. If you are doing something risky, then the `verbose` parameter doesn't provide protection against ill-advised actions like the `-whatif` or `=confirm` parameters, at least if you haven't worked out the precise effect of the command.

In this example, you will start three copies of the Notepad application and then use Windows PowerShell to stop those three instances but observe the output generated by Windows PowerShell when you use the `verbose` parameter. The example assumes that you don't have other instances of Notepad running.

Start three Notepad instances by typing these commands:

```
Notepad
Notepad
Notepad
```

Confirm that three Notepad processes are running:

```
get-process Notepad
```

Now stop the processes by typing:

```
stop-process -processname Notepad
```

You can see in Figure 8-17 that no information is given about what actions have been taken although all instances of Notepad are terminated. This will also close any other instances of Notepad you might have and will do so despite your perhaps having unsaved changes.

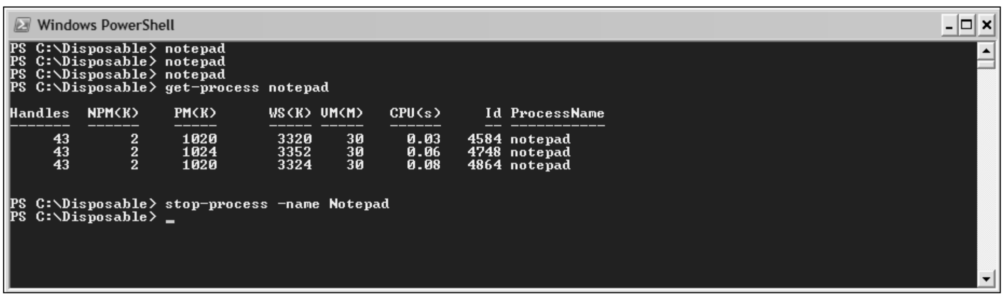


Figure 8-17

To observe the output Windows PowerShell generates with the `-verbose` parameter, start three new Notepad processes, as described above.

Then stop them using the `stop-process` command, but this time specifying the `verbose` parameter as follows:

```
stop-process -name Notepad -verbose
```

As you can see in Figure 8-18, all three instances of Notepad are terminated and information is given about each of the three Notepad processes the `stop-process` cmdlet has stopped.

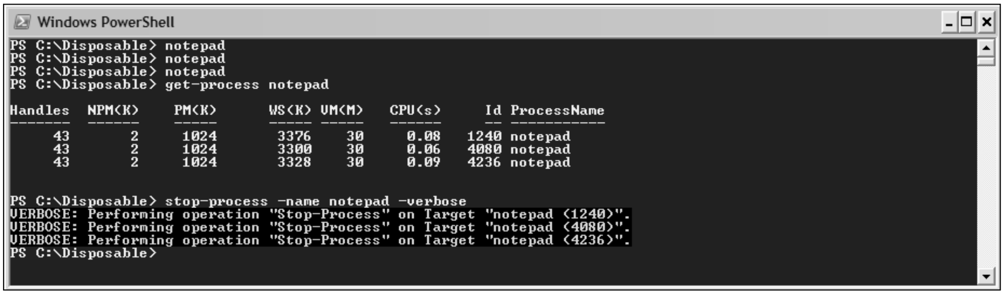


Figure 8-18

By specifying the `verbose` parameter, you get Windows PowerShell to display text information about each object affected by a command. This information is, by default, echoed to the console.

Summary

In this chapter, I described three parameters that are available on cmdlets that alter system state. The parameters provide either some protection against unintended changes in system state or feedback on changes made in system state.

The `-whatif` parameter allows you to test the effect of a command or pipeline without actually making any change in system state.

The `-confirm` parameter allows you to make a decision about each potential change in system state. You have options to proceed with or not to proceed with individual actions that affect system state.

The `-verbose` parameter provides additional information about the effects of a command, if the cmdlet supports the parameter.

In demonstrating the use of the preceding parameters, I introduced you to the following cmdlets, which can change system state:

- `remove-item`
- `stop-service`
- `stop-process`

and demonstrated how you could use them with the `-whatif`, `-confirm` and `-verbose` parameters.

9

Retrieving and Working with Data

Windows PowerShell allows you readily to access, retrieve, and manipulate data from a range of drives, files, and other data containers. Access to data stores in Windows PowerShell is founded on *providers*. A provider is a .NET program that makes available data from a data store and supports viewing and manipulation of that data.

In this chapter, I introduce you to several cmdlets that are relevant to exploring data stores and retrieving and manipulating data from them.

One of the differences between the Windows and the Linux families of operating systems is that Windows and Windows applications store system information in a huge variety of formats. In Linux, a lot of information is stored as text and, if you have the appropriate text utilities and the skills to use them, you can access a lot of system information by simply manipulating text. In Windows, you have system information stored in stores such as the registry and Active Directory. Text utilities just won't cut it. Windows PowerShell provides, and needs, a range of providers that allow you to access and manipulate objects that represent a variety of data stores.

Windows PowerShell Providers

Access to data stores in Windows PowerShell depends on providers. As mentioned earlier, a provider is a .NET program that allows you to access data in a data store, then display or manipulate it.

To find the providers available on a machine, use the `get-psprovider` cmdlet. To find all available providers, simply type:

```
get-psprovider
```

To display an alphabetical list of PowerShell providers, type:

```
get-psprovider | sort-object name
```

Figure 9-1 shows the names, capabilities and drives of the built-in PowerShell providers.

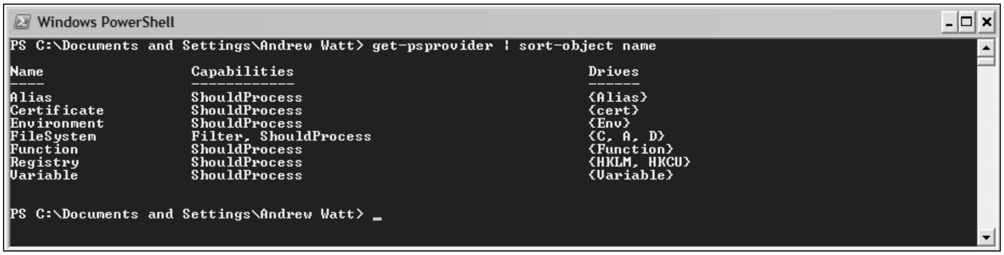


Figure 9-1

The following table summarizes the data stores supported by the built-in providers.

Provider	Data Store
`Alias`	Windows PowerShell aliases
`Certificate`	X509 certificates for digital signatures
`Environment`	Windows environment variables
`FileSystem`	File system drives, folders (directories) and files
`Function`	Windows PowerShell functions
`Registry`	Windows registry
`Variable`	Windows PowerShell variables

Broadly, each provider supports display of its data similar to how a traditional Windows command shell would display file system data. However, there are differences in detail. For example, the Alias provider does not (need to) support hierarchical data, since there is no concept of a folder of aliases.

Built-in Windows PowerShell providers are contained in snapins (which may also contain cmdlets). The built-in providers are contained in the Core and Security snapins that are loaded automatically by Windows PowerShell.

Using the get-psdrive Cmdlet

Windows PowerShell providers expose several different stores of information using the file system metaphor, but not all of these drives are conventional file system drives. You can demonstrate the variety of "drives" exposed by Windows PowerShell by typing the following command:

```
get-psdrive
```

The preceding command returns all current drives from all available providers. The `get psdrive` cmdlet has two parameters, which are implicit in the preceding command: `-name` and `-psprovider`, and the default value for both is `"*"`. So, the preceding command is actually equivalent to the command:

```
get-psdrive -name * -psprovider *
```

Figure 9-2 shows the results on a Windows XP machine.

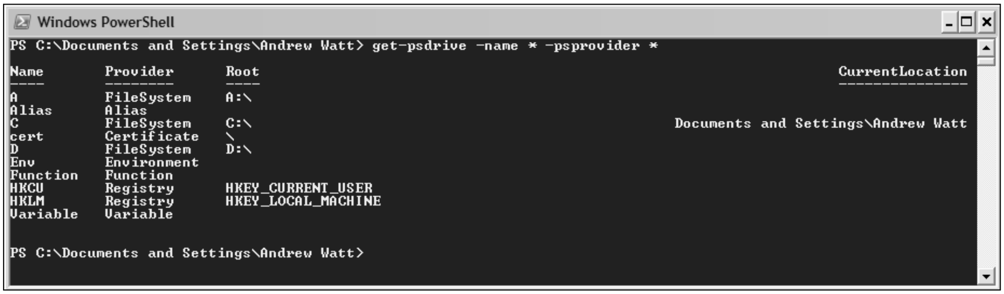


Figure 9-2

The `get-psdrive` cmdlet has four parameters (in addition to the common parameters covered in Chapter 6):

- `Name` — A positional parameter whose value is the name of a drive; it has a default value, the wildcard `*`.
- `Psprovider` — Specifies the provider. An optional named parameter; it has a default value, the wildcard `*`.
- `literalName` — Specifies a name for a drive that must be interpreted literally. In other words, any characters that are wildcards are treated as literal characters. An optional positional parameter; it cannot be used if the `-name` parameter is used.
- `Scope` — Specifies the scope. An optional named parameter.

For more information about named parameters, see Chapter 6.

The `get-psdrive` cmdlet supports the following providers, all in the `System.Management.Automation.Core` namespace, in Windows PowerShell version 1:

- `FileSystem` — Exposes information about conventional file system drives
- `Alias` — Exposes information about aliases available in the current system state
- `Certificate` — Exposes information about certificates on the current machine
- `Environment` — Exposes information about the environment variables on the current machine
- `Function` — Exposes information about functions in the current system state
- `Registry` — Exposes information about selected hives in the registry
- `Variable` — Exposes information about variables in the current system state

To see which drives are available from a specified provider, you supply a value for the `-psprovider` parameter. For example, to see the drives returned by the `Registry` provider, type the following command:

```
get-psdrive -psprovider Registry
```

Figure 9-3 shows the result.

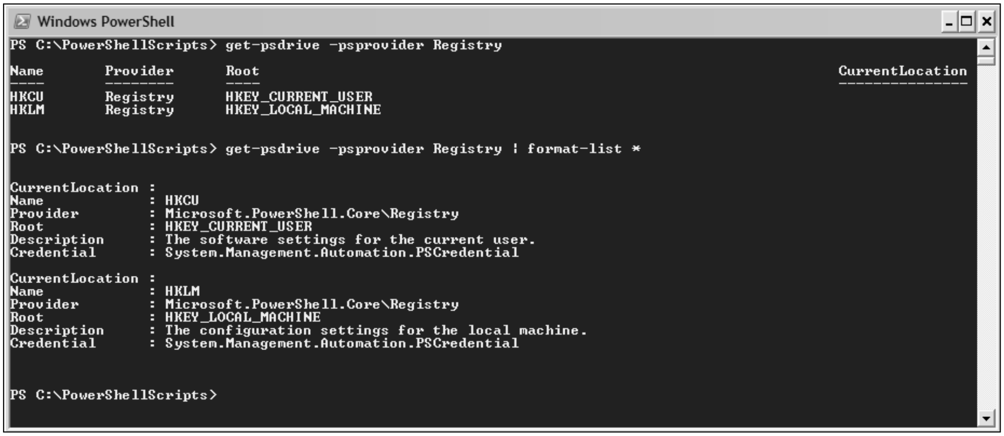


Figure 9-3

You can use the `format-list` cmdlet, which I introduced in Chapter 7, to display the information in a list format. As you can see in the lower part of Figure 9-3, this causes additional information about each drive to be displayed.

To see a convenient display of which drives on a machine are associated with which provider, use this command:

```
get-psdrive |
select-object Name, Provider |
group-object provider |
format-list
```

As you can see in Figure 9-4, only two providers (`FileSystem` and `Registry`) by default expose more than one drive. The `get-psdrive` cmdlet returns drives, on this particular machine, which include file store drives that represent a conventional floppy drive, a hard drive, a CD/DVD drive, and two registry drives: HKCU (Current User) and HKLM (Local Machine), which represent the correspondingly named registry hives.

To explore what is in the drives that the `get-psdrive` cmdlet returns, use the `get-childitem` cmdlet described later in this chapter.

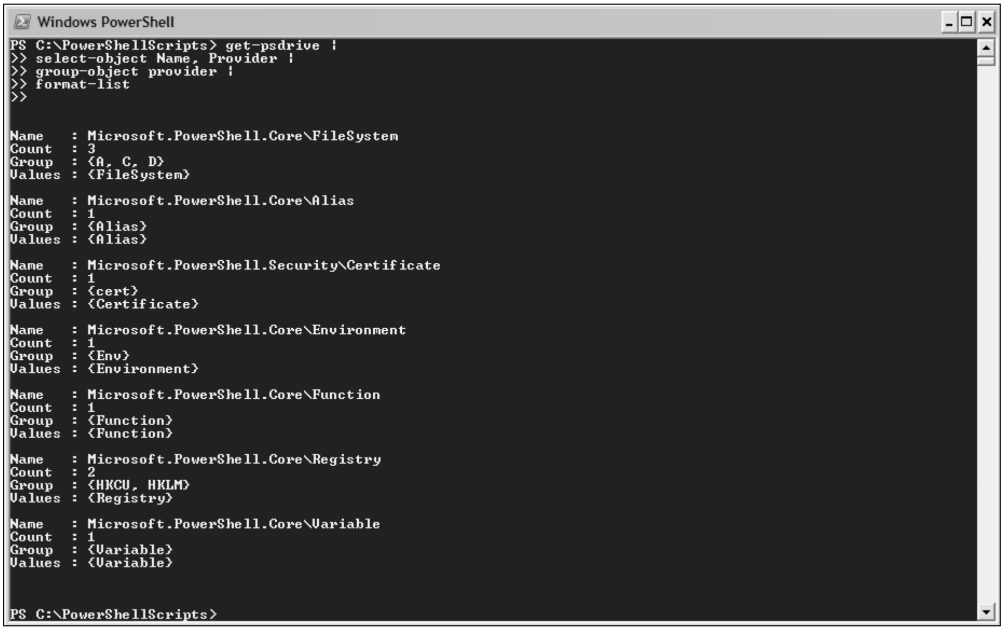


Figure 9-4

The `remove-psdrive` cmdlet deletes a PowerShell drive. In addition to the common parameters, the `remove-psdrive` cmdlet supports the following parameters:

- `name` — Specifies the name(s) of the PowerShell drive(s) to be removed.
- `psprovider` — Specifies which PowerShell providers the drives to be removed belong to.
- `scope` — An index used to identify the scope.
- `force` — Allows the cmdlet to override nonsecurity restrictions.
- `whatif` — Describes what would happen if you executed the command. No change is actually made.
- `confirm` — Specifies that PowerShell should prompt for confirmation before executing the command.

Suppose that you had created a drive called `Scripts` whose root folder is located at `C:\PowerShellScripts`, using the `new-psdrive` cmdlet:

```
new-psdrive -name Scripts -psProvider FileSystem -root C:\PowerShellScripts
```

To remove that PowerShell drive, you use the `remove-psdrive` cmdlet:

```
remove-psdrive -name Scripts -psProvider FileSystem
```

To demonstrate that you have successfully removed the Scripts drive, use this command:

```
set-location Scripts:
```

You will see the following error message, which indicates that the `Scripts` drive no longer exists on the system:

```
Set-Location : Cannot find drive. A drive with name 'Scripts' does not exist.
At line:1 char:3
+ cd  <<<< Scripts:
```

Using the set-location Cmdlet

Once you find supported drives with Windows PowerShell, you will likely want to navigate around in them. You use the `set-location` cmdlet to do that. To simplify the use of providers and drives, Windows PowerShell defines several aliases.

To find the aliases for `set-location` on your system, type this command:

```
get-alias |
where-object {$_.Definition -eq "set-location"}
```

The results are shown in Figure 9-5. Whether you use `sl`, `cd` or `chdir` is your choice.

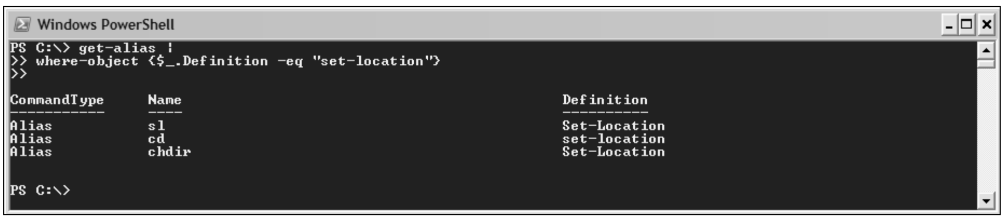


Figure 9-5

The `get-alias` cmdlet with no parameters returns objects representing all aliases on the current system. The `where-object` cmdlet in the second step of the pipeline filters those so that only those whose `Definition` property has the value of `set-location` are passed on to the default formatter.

The `set-location` cmdlet supports the following parameters in addition to the common parameters:

- `path` — Specifies the path for the new working location. This is a positional parameter in position 1. The default is the empty string.
- `literalPath` — Specifies a path. The value of this parameter is to be interpreted literally. In other words, any wildcard characters in the path are treated literally.
- `passthru` — Specifies that the object created by the cmdlet is to be passed along the pipeline.
- `stackname` — Specifies the stack to which the location is being set. If no value is specified, then the current working stack is used. A stack is a data structure based on the last in, first out principle.

To set the location to the root directory of the C: drive from any location, simply type:

```
set-location c:\
```

or:

```
set-location -path c:\
```

The preceding command works whether you are in a drive supported by the `FileSystem` provider or in a location supported by another provider, for example, `Registry` or `Certificate`.

If your system supports the `cd` alias, you can do the same thing by typing:

```
cd c:\
```

If you don't want to change drives (in other words, you want to change folder on a drive), you don't need to provide the drive letter. For example, if you want to stay on the current `FileSystem` drive and set the location to the root directory, simply type:

```
set-location \
```

or, using the `cd` alias:

```
cd \
```

If you want to navigate to a location whose name includes spaces, you must enclose the value specified for the `-path` parameter in paired quotation marks, or format it as a variable containing the full name. For example, to navigate to `C:\Documents and Settings\Administrator`, you must type:

```
set-location "C:\Documents and Settings\Administrator"
```

or:

```
$loc= "C:\Documents and Settings\Administrator"
set-location $loc
```

If you omit the quotation marks when specifying a location name containing one or more spaces, an error message is displayed.

You can use the `..` abbreviation for a parent. You can navigate across multiple levels. For example, to move from `C:\Documents and Settings\Administrator` to the root directory, type this command (which moves you to the parent of a parent):

```
set-location ..\..
```

Figure 9-6 shows the results from the two preceding commands.

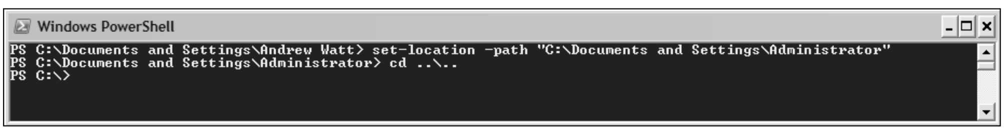


Figure 9-6

If your system has a `c:` function specified in the relevant profile files, you may only need to type

```
c:
```

to switch from any drive to the previously current location on drive c:.

Using the passthru Parameter

The normal behavior of the `set-location` cmdlet is to set a new location. By using the `-passthru` parameter, you cause the `set-location` cmdlet to pass objects to later steps in a pipeline.

The following command switches to the environment variable drive (`env:`):

```
set-location env:
```

The result is shown in the upper part of Figure 9-7.

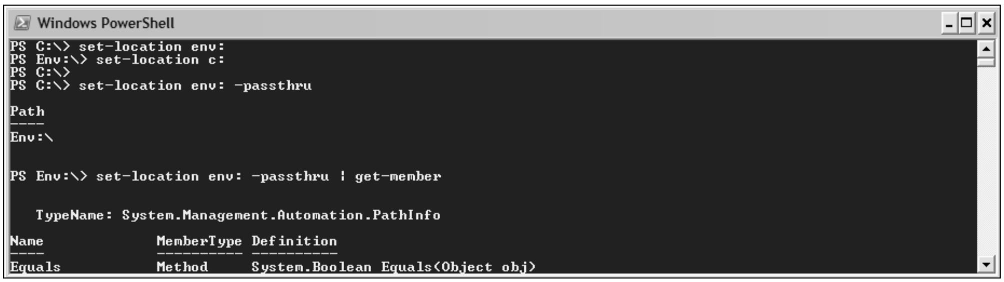


Figure 9-7

Notice that nothing is passed to the default formatter, so the prompt for the `env:` drive is simply displayed. However, if you use the `-passthru` parameter, the `set-location` cmdlet creates a `PathInfo` object and passes it to the pipeline and to the default formatter:

```
set-location env: -passthru
```

You can confirm that a `PathInfo` object has been created by using the `get-member` cmdlet, as in the following command:

```
set-location env: -passthru | get-member
```

A `PathInfo` object relating to the environmental variable drive is created and passed to the default formatter with the result shown in the Figure 9-7.

> **The `get-location`, `push-location` and `pop-location` cmdlets are described in Chapter 15.**

Using the get-childitem Cmdlet

The `get-psdrive` cmdlet returns information about drives on a machine or that you created, and the `set-location` cmdlet lets you switch between various drives and between folders within drives. The `get-childitem` cmdlet allows you to retrieve information about the items in a folder.

> *The behavior of the* `get-childitem` *cmdlet has been the subject of considerable discussion during PowerShell's development. Some users, including myself, found the semantics surprising in some situations. The behavior described in this section is that found in the final release version of PowerShell 1.0.*

Windows PowerShell typically supports aliases for the `get-childitem` cmdlet. You can find those aliases on your machine by using the following command:

```
get-alias |
where-object {$_.Definition -eq "get-childitem"}
```

In a default install, the aliases `dir`, `ls`, and `gci` are likely to be available to you.

In addition to the common parameters, the `get-childitem` cmdlet supports the use of a number of parameters. However, since the `get-childitem` cmdlet can be used with a range of providers, not all of the supported parameters work with all providers. The supported parameters are:

- `path` — A positional parameter, in position 1, which specifies the location relative to which the child items are to be found. If no value is supplied, the default is the current location. It cannot be used with the `-literalPath` parameter.
- `literalpath` — A positional parameter, in position 1, whose value is to be interpreted literally. It cannot be used with the `-path` parameter.
- `include` — Filters in items among those specified by the value of the `-path` parameter.
- `exclude` — Filters out items among those specified by the value of the `-path` parameter.
- `filter` — Specifies filter elements as required and supported by providers.
- `name` — A boolean-valued parameter. If `$true`, then only the name of an item is streamed. The default value is `$false`, in which case the item (not just its name) is streamed.
- `recurse` — A boolean-valued parameter that specifies whether or not folders are to be searched recursively.
- `force` — A boolean-valued parameter that may expose additional items, for example hidden files in a directory. The cmdlet should still respect any security settings on a folder.

In Chapter 8, you created a directory called `Disposable` and populated it and a subfolder with some test files. You need these structures for the following examples.

To find the items contained in the `C:\Disposable` folder from any location, you can type the following command:

```
get-childitem -path C:\Disposable
```

The value of the `path` parameter specifies the location relative to which the child items are to be found, in this case the `C:\Disposable` folder. Figure 9-8 shows the result.

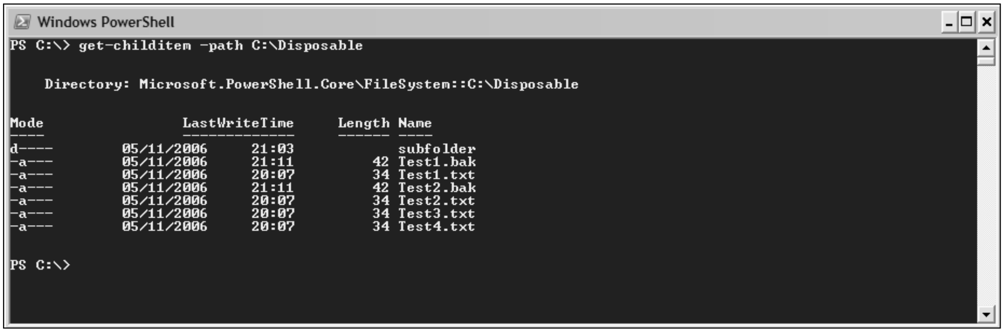


Figure 9-8

The `-include` parameter filters items specified by the `-path` parameter. In the following example, move to the `Disposable` folder using the command

```
set-location C:\Disposable
```

modified as appropriate for your machine, if you used a drive other than C:. Then use the `*` wildcard to retrieve all child items of the `Disposable` folder:

```
get-childitem *
```

That retrieves all files and folders in the `Disposable` directory, as shown in the upper part of Figure 9-9.

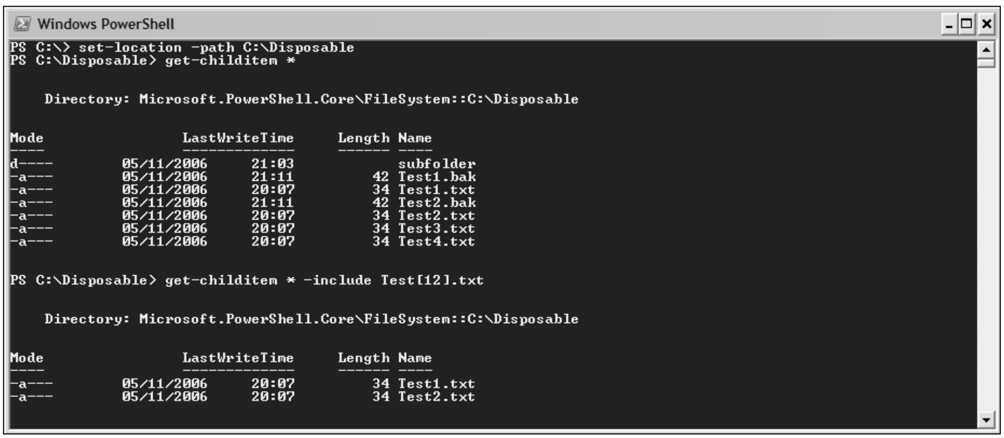


Figure 9-9

When you add the `include` parameter with a value of `Test[12].txt`:

```
get-childitem * -include Test[12].txt
```

only the files `Test1.txt` and `Test2.txt` are returned, as shown in the lower part of Figure 9-9. In this example, the value supplied for the `-include` parameter uses a regular expression. With regular expressions, the class, `[12]` matches any of the characters contained in it, that is the numeric digits 1 and 2. Thus, `Test[12].txt` matches only `Test1.txt` and `Test2.txt`.

However, the `-include` parameter can occasionally produce surprising results for a parameter intended to filter results. If you modify the command to

```
get-childitem * -include *
```

the "filtered" children are more than the original results (which were shown in Figure 9-9), as shown in the lower part of Figure 9-10. Not only do you see the children of `C:\Disposable`, but you also see the children of the `C:\Disposable\subfolder` folder.

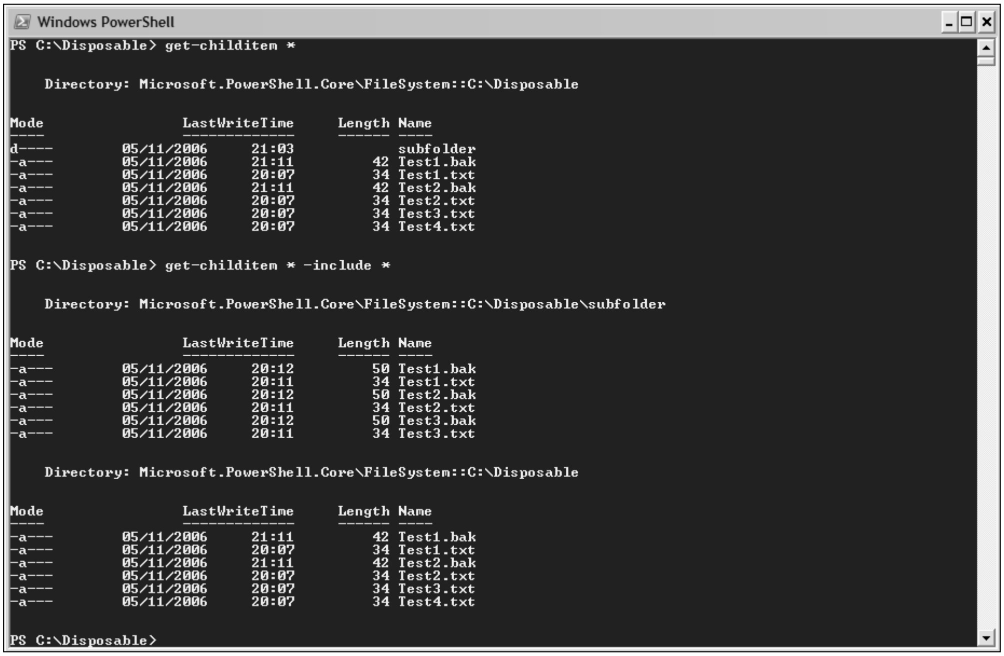


Figure 9-10

Windows PowerShell interprets the * as matching any file or folder in the current directory. It then finds any "child" of those files and folders. Windows PowerShell treats a file as being its own child. So, while all the files in the `Disposable` directory are displayed. The `subfolder` folder's children are also found.

As another example of the potentially counterintuitive behavior of the `get-childitem` cmdlet consider the following command:

```
get-childitem -path .
```

indicates that the children of the current location are to be found. As you can see in Figure 9-11, all the children of the `C:\Disposable` folder are listed.

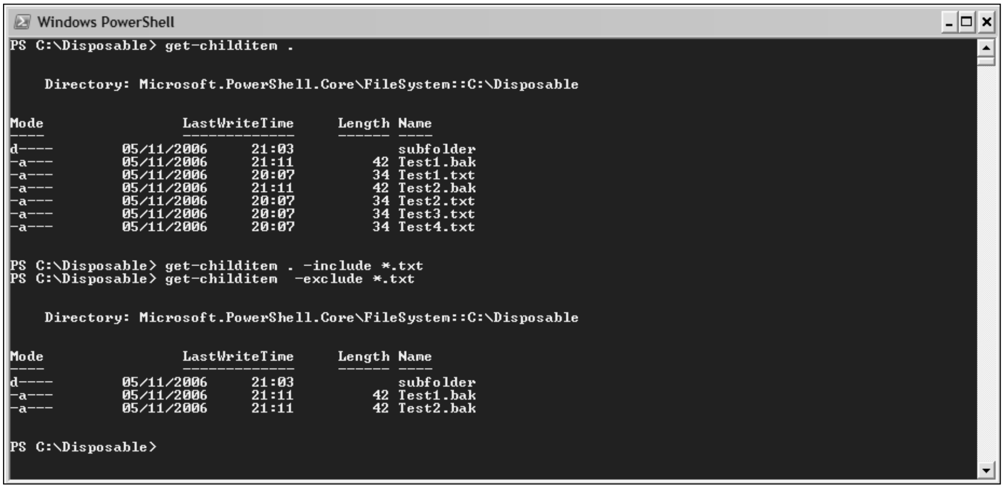


Figure 9-11

However, if you add the `-include` parameter, as in the following command:

```
get-childitem -path . -include *.txt
```

no children are displayed, as you can see in the middle section of Figure 9-11. If the `-include` parameter is to behave as a filter I expected the `.txt` files to be displayed. However, if you use the `-exclude` parameter, as in the following command:

```
get-childitem -path . -exclude *.txt
```

it behaves as I expected.

Suffice it to say, that the behavior of the `get-childitem` cmdlet and its parameters in the preceding examples (and others) can cause significant confusion to some users. In the light of that I suggest you exercise significant care if you use the `get-childitem` cmdlet to pipe objects to later pipeline steps. As the preceding examples indicate, you may not be piping what you expect.

Using the get-location Cmdlet

The `get-location` cmdlet returns the current location for a specified provider or, if no provider is specified, for the current provider. If you use the current location as part of the Windows PowerShell prompt when using a `FileSystem` provider drive, then the `get-location` cmdlet doesn't tell you much that you don't already know. However, if you use an alternate prompt, such as the current date and time, the `get-location` cmdlet lets you know which directory you are currently working in.

The following parameters are available for use with the `get-location` cmdlet, in addition to the common parameters. All parameters of the `get-location` cmdlet are named parameters.

- `psprovider` — Specifies a provider (or providers)
- `psdrive` — Specifies a drive (or drives)
- `stack` — If present, specifies that the item is to be taken from the current stack
- `stackname` — Specifies a stack from which items are to be retrieved

The `-psprovider` and `-psdrive` parameters can be used together. Similarly, the `-stack` and `-stackname` parameters can be used together.

The straightforward command

```
get-location
```

retrieves the current location for the current provider and displays the result as a full path, as shown in Figure 9-12.

Figure 9-12

If you want to find the current working location in any provider, you can explicitly specify the name of the provider of interest using the `-psprovider` parameter. For example, if you are working in the file system and want to find out your most recent location in the registry, use the following command:

```
get-location -psprovider Registry
```

Figure 9-13 shows the result.

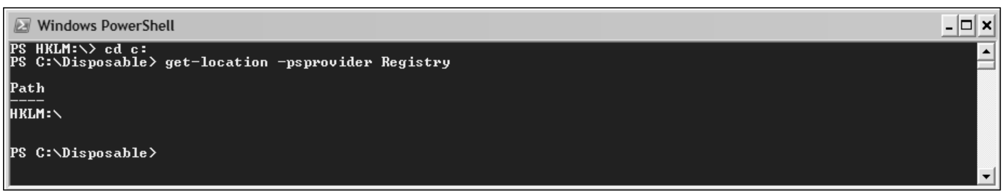


Figure 9-13

Some providers support the ability to have a current location on multiple drives. For example, using the `FileSystem` provider, you can have, for example, multiple drives, such as C: and D:. You have a current location on each drive. When you start a PowerShell console, the current location on each drive is the root folder, except on the system drive, where the folder is specified in the variable `$home`. For me, that is the folder `C:\Documents and Settings\Andrew Watt`.

To find your location on drive D, use the following command:

```
get-location -psprovider FileSystem -psdrive D
```

As shown in the upper part of Figure 9-14, the current location on drive D: using the `FileSystem` provider is returned.

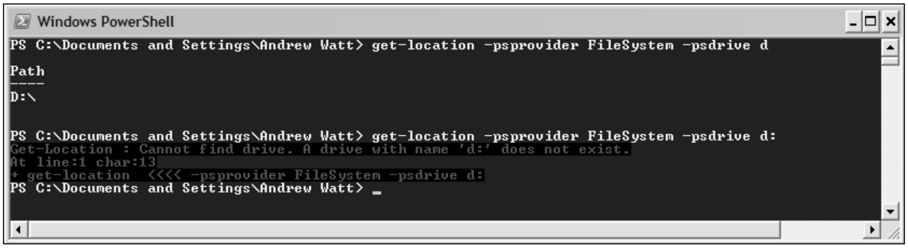


Figure 9-14

Be careful not to include the colon in the drive name. As shown in the lower part of Figure 9-14, including the colon in the drive name produces an error message:

```
get-location -psprovider FileSystem -psdrive D:
```

> **The `-stack` and `-stackname` parameters are used to retrieve items from the default stack or a named stack, in situations where you use the `push-location` and `pop-location` cmdlets. These cmdlets are described in Chapter 15.**

Using the get-content Cmdlet

The `get-content` cmdlet returns the content of a specified item at a specified location. You are likely to have aliases available to use the `get-content` cmdlet, for example `cat`, `type`, and `gc`. If you are unsure which aliases are available on your system use, the following command to find out:

```
get-alias |
where-object {$_.definition -eq "get-content"}
```

The Windows PowerShell help files occasionally differ in completeness or accuracy from the actual implementation of a cmdlet. A useful technique to find the current functionality is to use the `definition` *property. For example, to retrieve the information for the* `get-content` *cmdlet, use this command:* `(get-command get-content).definition.`

The `get-content` cmdlet supports the following parameters, in addition to the common parameters.

- `Path` — Specifies the path to the item or file that data is to be retrieved from.
- `Readcount` — Specifies how many lines are to be sent through the pipeline at a time. The default value is 0 (all lines).

- `Totalcount` — Specifies the number of elements (often lines) to retrieve from the target file or item. The default value is `-1` (retrieve all lines).
- `Filter` — Specifies filter elements as required and supported by providers.
- `Include` — Specifies which container's items are to be retrieved. The value of this parameter qualifies the value of the `-path` parameter.
- `Exclude` — Specifies which container's items are not to be retrieved. The value of this parameter qualifies the value of the `-path` parameter.
- `Force` — Allows, subject to security, overriding of some constraints such as file renaming.
- `Credential` — Specifies a credential to authenticate access.
- `Delimiter` — Specifies an alternative delimiter between "lines."
- `Wait` — A boolean parameter. If `$true` then a watch is kept on the item from which elements are being retrieved, so that if the item is updated the newly added item is retrieved.
- `Encoding` — Specifies the character encoding to be used to display the content.

In the following examples I use a simple text file, `TenLines.txt` to demonstrate the use of `get-content`. The file contains the following content, saved to the root directory of drive C.

```
This is line 1.
This is line 2.
This is the third line.
This is the fourth line.
This is line 5.
This is line 6.
This is line 7.
This is line 8.
This is line 9.
This is line 10.
```

Simple use of the `get-content` cmdlet specifies a single file from which to retrieve data. For example, to retrieve all the data in the file `C:\TenLines.txt`, type the following command:

```
get-content -path C:\TenLines.txt
```

As you can see in Figure 9-15, all the content of the file is retrieved and displayed on the console.

Figure 9-15

To limit the number of lines retrieved from a file use the `-totalcount` parameter. Its value is an integer. The following command limits the number of lines retrieved to 5.

```
get-content -path C:\TenLines.txt -totalcount 5
```

The result is shown in Figure 9-16.

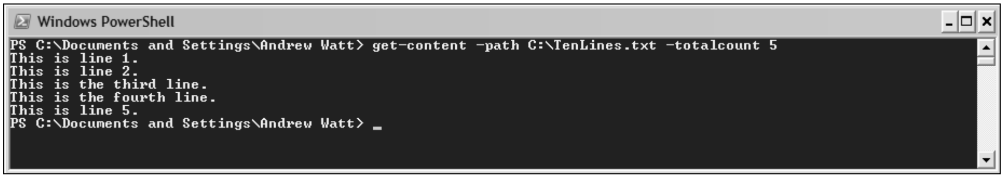


Figure 9-16

Be careful if you use the `-delimiter` parameter. The delimiter is included in each "line." Suppose that you have a simple file, `Delimiter.txt`, with the following content:

```
and;bcd;ckk
```

To split the text into "lines" at each semicolon use the following command:

```
$chunks = get-content Delimiter.txt -delimiter ";"
```

The variable `$chunks` is an array with the following content:

```
and;
bcd;
ckk
```

Notice that the semicolon character is included in the first two elements of the array.

The `get-content` cmdlet can be combined with other cmdlets in a pipeline. For example, using the `measure-object` cmdlet you can display information such as counts of lines, words, or characters in a text file. I discuss the `measure-object` cmdlet in the next section.

To do a count of lines, words, and characters in the `C:\TenLines.txt` file, use the following command:

```
get-content -path C:\TenLines.txt |
measure-object -line -word -character |
format-table Lines, Words, Characters -auto
```

Figure 9-17 shows the result. If you don't specify a step using the `format-table` cmdlet, the default formatter displays a blank `Property` column. By specifying specific columns to be displayed, the appearance is a little tidier. I discuss formatting of output in Chapter 7.

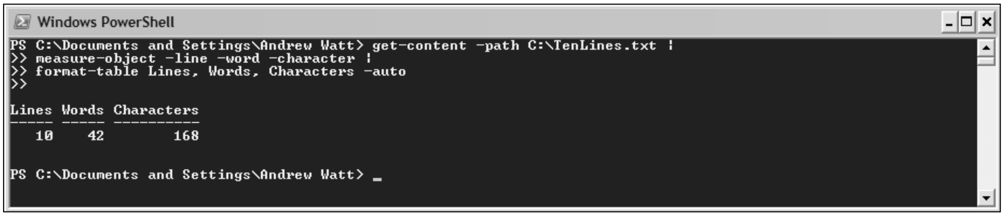


Figure 9-17

The -wait parameter allows you to keep an eye on any new content being added to a file. To run this example, you need to have two Windows PowerShell windows open.

First, redirect a literal string to create a file C:\Content.txt:

```
"Hello world!" > C:\Content.txt
```

then using the get-content cmdlet, confirm that the file has been created and the text successfully added using this command:

```
get-content -path C:\Content.txt
```

Next, add another line to Content.txt. To append text, you use the >> operator.

```
"This is a second line." >> C:\Content.txt
```

Then confirm that the second line of text has been appended using:

```
get-content -path C:\Content.txt
```

Figure 9-18 shows the appearance in the data entry Windows PowerShell window after the four preceding steps.

Figure 9-18

Now switch to the data-monitoring Windows PowerShell window. First use the get-content cmdlet with only the -path parameter to confirm the content and also to demonstrate that the prompt is displayed immediately after the file's content is displayed.

```
get-content -path C:\Content.txt
```

Then add a `-wait` parameter:

```
get-content -path C:\Content.txt -wait
```

Notice that the same text is in the file but the prompt is not displayed. Instead, the cursor flashes on a blank line. Figure 9-19 shows the situation in the data-monitoring window after the preceding two steps.

Figure 9-19

Switch to the data entry window and add two more lines to `Content.txt` using the following commands. Notice that after each command is executed, the result displayed in the data-monitoring window is updated to reflect the updated content `C:\Content.txt`.

```
"This is a THIRD line." >> C:\Content.txt
"This is a FOURTH line." >> C:\Content.txt
```

Figure 9-20 shows the data-monitoring window after the execution of the two preceding commands. There appears to be a bug when displaying data added while the `-wait` parameter is in operation.

Figure 9-20

The `-include` parameter specifies from which container (specified in the `path` parameter) items are to be retrieved.

The following example uses the files in the `C:\Disposable` directory used earlier in the chapter. To retrieve the content of each of the `.txt` files in the folder, you can use the `include` parameter, as in this command:

```
get-content -path C:\Disposable\* -include *.txt
```

To filter the files further, you can use wildcards in the value of the `include` parameter. For example to retrieve the content of only `Test2.txt` and `Test3.txt`, use the following command:

```
get-content -path C:\Disposable\* -include *[23].txt
```

Figure 9-21 shows the result of running the two preceding commands.

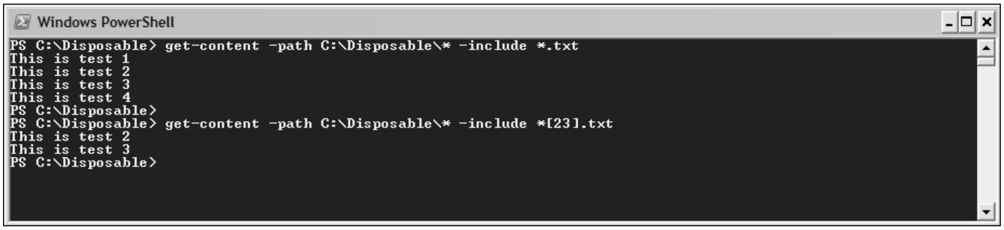


Figure 9-21

Using the measure-object Cmdlet

The `measure-object` cmdlet allows you to measure or calculate properties of Windows PowerShell objects. One use of the `measure-object` cmdlet is to provide information on summary criteria, such as line count or word count, on files whose content is retrieved using the `get-content` cmdlet.

In addition to the common parameters, the `measure-object` cmdlet supports the following parameters:

- `InputObject` — Specifies the input object. If the input comes from a pipeline, this parameter is omitted.
- `Property` — Specifies a property on which the cmdlet is to operate.
- `Average` — Specifies that the mean of some numeric items is to be calculated.
- `Sum` — Specifies that the sum of numeric values is to be calculated.
- `Minimum` — Specifies that the minimum value in a series of numeric values is to be found.
- `Maximum` — Specifies that the maximum value in a series of numeric values is to be found.
- `Line` — Specifies that a line count of text data is to be carried out.
- `Word` — Specifies that a word count of text data is to be carried out.
- `Character` — Specifies that characters in text data are to be counted.
- `IgnoreWhitespace` — A boolean value that specifies whether or not whitespace is to be ignored. By default, whitespace characters are counted.

The `-inputObject` and `-property` parameters apply to numeric and text input. The `-average`, `-sum`, `-minimum`, and `-maximum` parameters are used with numeric input. The `-line`, `-word`, `-character`, and `IgnoreWhitespace` parameters are used with text input.

The following code calculates the average, sum, minimum, and maximum length of the files in a folder. Objects are supplied to the `measure-object` cmdlet via the pipeline:

```
get-childitem * |
measure-object -property Length -Average -Sum -Minimum -Maximum |
format-table Count, Average, Sum, Minimum, Maximum -auto
```

Figure 9-22 shows the results. The `-property` parameter specifies that the `Length` property of the child items is the basis for the calculations. Notice that the `-average`, `-sum`, `-minimum`, and `-maximum` parameters are used. The count of the items is calculated automatically. The columns specified in the `format-table` statement avoid the display of a repeated `Property` column (which the default formatter would otherwise create).

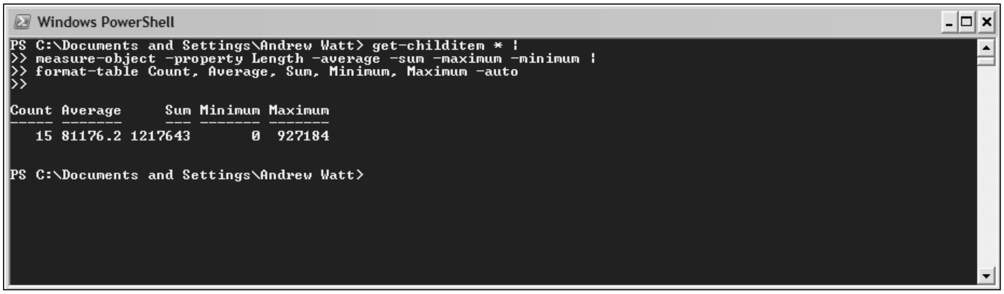


Figure 9-22

You can also use the `measure-object` cmdlet with the `get-childitem` cmdlet to count the number of files and subfolders in a folder. To find the number of files and folders in `C:\Windows\System32` use this command:

```
get-childitem -path C:\Windows\System32\* |
measure-object
```

The `*` wildcard at the end of the path parameter matches the names on all files and subfolders. If you modify the wildcard to `*.*`, as in the following command, only items whose name includes characters followed by a period followed by characters are counted.

```
get-childitem -path C:\Windows\System32\*.* |
measure-object
```

Figure 9-23 shows the results on one Windows XP machine.

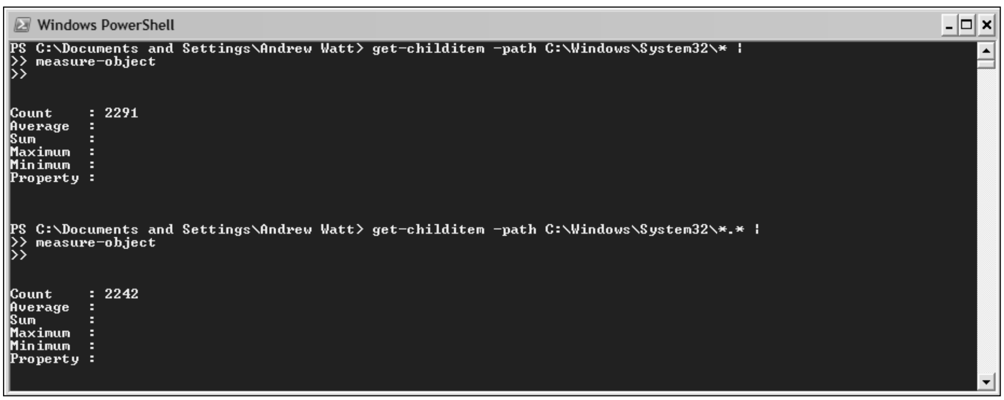


Figure 9-23

The new-item Cmdlet

The `new-item` cmdlet allows you to create a new item in a namespace. In addition to the common parameters, the `new-item` cmdlet supports the following parameters:

- `path` — Specifies the path to the new item
- name — Specifies the name of the new item
- `itemtype` — Specifies the type of the new item (varies by provider)
- `value` — Specifies, if appropriate, a value for the new item
- `force` — Allowing for security, may override other constraints
- `credential` — Specifies a credential for the action
- `whatif` — Shows the user the potential result, but no change is made
- `confirm` — Prompts the user to confirm whether or not a new item or items should be created

To use the `new-item` cmdlet to create a new text file, `C:\Test2.txt`, follow these steps:

First, check the text file content of the `C:\ directory`:

```
get-childitem -path C:\T*.txt
```

Next, use `new-item` to create a new item `C:\Test2.txt` and specify that it is a file:

```
new-item -path C:\Test2.txt -type file
```

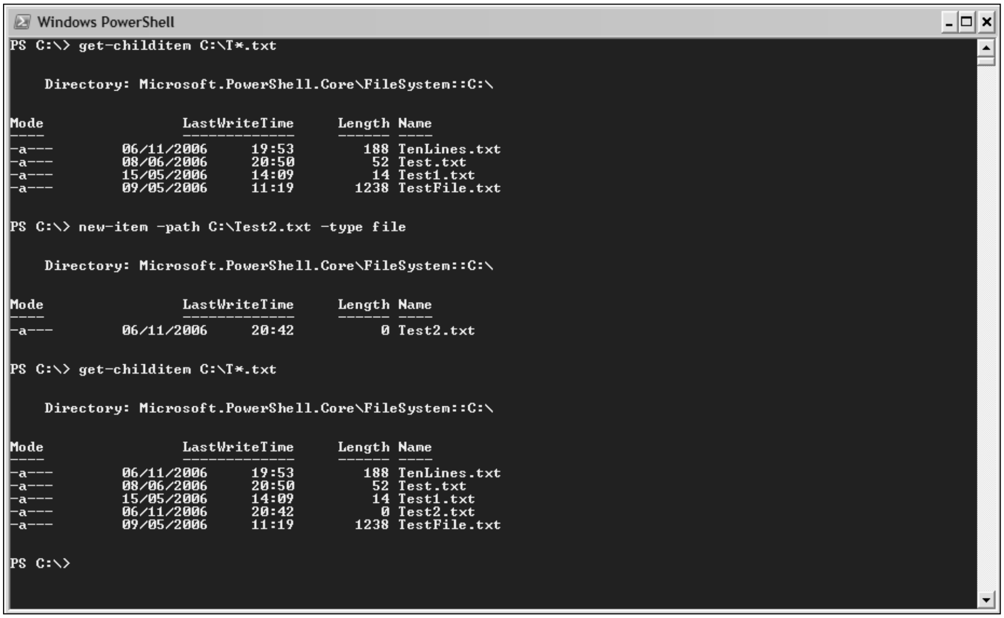
Then confirm that a new file has been added:

```
get-childitem -path C:\T*.txt
```

Figure 9-24 shows the before and after results.

To add a new folder named `C:\TestFolder`, use this command:

```
new-item -path C:\TestFolder -type directory
```



```
Windows PowerShell
PS C:\> get-childitem C:\T*.txt

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        06/11/2006     19:53        188 TenLines.txt
-a---        08/06/2006     20:50         52 Test.txt
-a---        15/05/2006     14:09         14 Test1.txt
-a---        09/05/2006     11:19       1238 TestFile.txt

PS C:\> new-item -path C:\Test2.txt -type file

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        06/11/2006     20:42          0 Test2.txt

PS C:\> get-childitem C:\T*.txt

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        06/11/2006     19:53        188 TenLines.txt
-a---        08/06/2006     20:50         52 Test.txt
-a---        15/05/2006     14:09         14 Test1.txt
-a---        06/11/2006     20:42          0 Test2.txt
-a---        09/05/2006     11:19       1238 TestFile.txt

PS C:\>
```

Figure 9-24

The new-psdrive Cmdlet

The `new-psdrive` cmdlet allows you to create a custom drive. For example, you might want to create a new drive called Scripts, which is located at `C:\PowerShellScripts`. This facility allows you convenient access to folders which might require tedious typing.

In addition to the common parameters, the `new-drive` cmdlet supports the following parameters:

- `Name` — Specifies the name of the custom drive
- `PSprovider` — Specifies which provider is to be used to create the drive
- `Root` — Specifies the location of the root of the custom drive
- `Description` — Provides a description of the drive's use or purpose
- `Scope` — Specifies the scope for the new drive
- `Credential` — Specifies the credential supplied to obtain any necessary authorization to create the new drive
- `Whatif` – Shows the user the potential result, but no change is made
- `Confirm` — The user is asked to confirm whether or not a new drive should be created

To create a new drive named `Writing` that allows easy access to the `C:\My Writing` folder, which already exists on one of my machines, use this command:

```
new-psdrive -Name Writing -psProvider FileSystem -Root "C:\My Writing"
```

To switch to the newly created `Writing` drive use this command:

```
cd Writing:
```

In this situation, you must provide the colon with the drive name or you will get an error message.

To find the information about this book, I typed this command:

```
get-childitem *PowerShell*
```

Figure 9-25 shows the results on the machine in question.

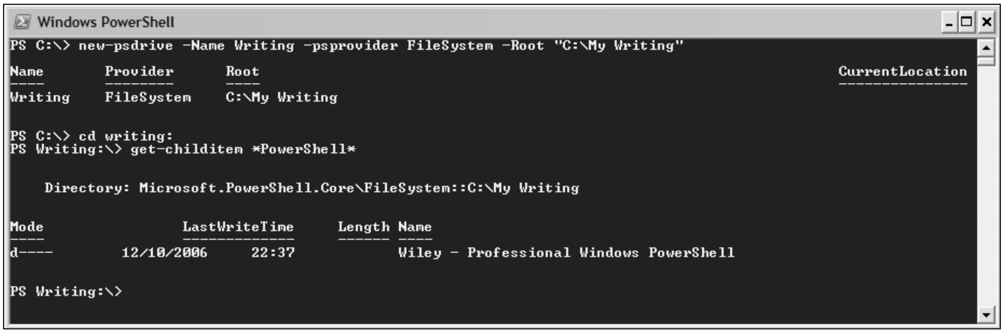


Figure 9-25

Summary

Access to data stores in Windows PowerShell is built on *providers*. Providers are .NET programs that allow PowerShell users to access data in a data store and present that data in a way similar to a file system.

I introduced you to several cmdlets that allow you to access data when using Windows PowerShell or navigate a data store:

- ❑ `get-psdrive` — Finds what PowerShell drives are available
- ❑ `set-location` — Sets a location
- ❑ `get-childitem` — Retrieves information about child items of a specified location
- ❑ `get-content` — Retrieves content from a file
- ❑ `measure-object` — Allows you to calculate and display summary information about, for example, a file
- ❑ `new-item` — Allows you to create a new item, for example, a file or folder
- ❑ `new-psdrive` — Allows you to create a new custom drive

10

Scripting with Windows PowerShell

In the preceding chapters, I illustrated the functionality of individual cmdlets but put little emphasis on using Windows PowerShell as a scripting language. In this chapter, I introduce several aspects of the Windows PowerShell language that make it suitable for scripting, and describe and demonstrate how many of its various components can be used. This foundational understanding of the PowerShell scripting language, taken together with your understanding of the various cmdlets, built up in the preceding chapters, will give you the knowledge necessary to explore in the following chapters the range of ways that Windows PowerShell can be used.

Chapter 11 introduces several more features of the Windows PowerShell language.

Enabling Scripts on Your Machine

At the time of this writing, the default configuration of Windows PowerShell when it is installed doesn't allow you to run scripts. If your local administrator has enabled scripts on your machine, then you may not need to take the steps described later in this section.

Windows PowerShell supports four *execution policies*, listed here. An execution policy determines whether you can run PowerShell scripts at all and which scripts you can run. The `Restricted` execution policy is the default. The four execution policies supported in Windows PowerShell 1.0 are:

- ❑ `Restricted` — Windows PowerShell operates as an interactive shell only. You cannot run any `.ps1` scripts or `.ps1xml` configuration files at startup.
- ❑ `AllSigned` — Runs only scripts that have first been signed by a publisher that you trust. This includes scripts that you create on the local computer.

- `RemoteSigned` — Windows PowerShell runs locally authored scripts that are not digitally signed, but any scripts downloaded from applications like Internet Explorer and Microsoft Outlook must be signed by a publisher that you trust before you can run them.
- `Unrestricted` — PowerShell runs all scripts. Scripts downloaded from applications like Internet Explorer display a prompt that indicates that they have been downloaded.

If you attempt to run a script where the execution policy is `Restricted` and therefore forbids its execution, you will see an error message similar to the one in Figure 10-1.

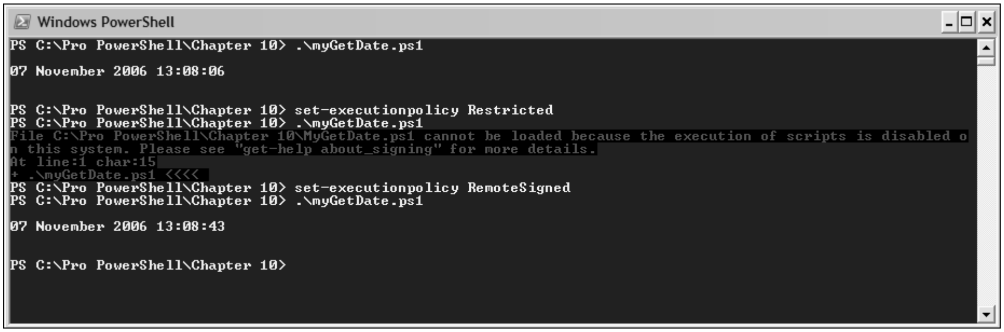


Figure 10-1

If you attempt to execute an unsigned script when the execution policy is AllSigned, you will see an error message similar to the one in Figure 10-2.

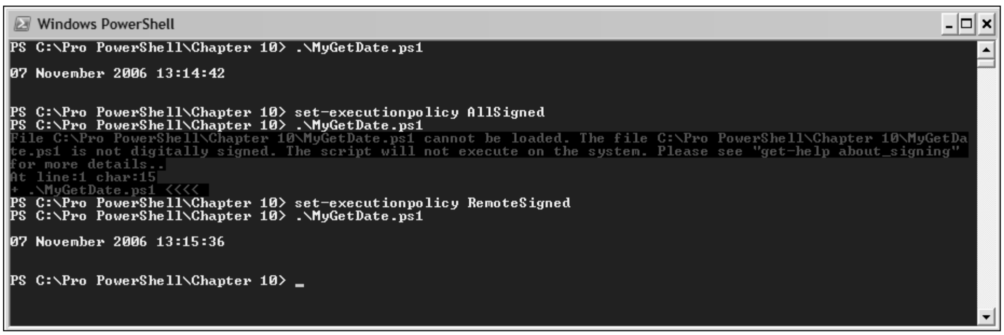


Figure 10-2

The four execution policies cover a spectrum of increasingly easy access to running Windows PowerShell scripts, which may, of course, have both good and bad points. In a testing scenario if you know that you won't download any malicious scripts, then `Unrestricted` is very convenient. The examples in this chapter assume that you have set the execution policy to `Unrestricted`—*for a development and test machine only!* You can, of course, sign any scripts and run them if that is required by an `AllSigned` execution policy. If there is a likelihood that you may run downloaded scripts, then the `RemoteSigned` policy gives you a little more protection.

> **If you download seemingly useful Windows PowerShell code from the Internet, be sure that you understand what it does before you copy and paste it into your own scripts. If you create and save the script, even if it includes malicious code that you have copied and pasted into it, then you bypass the protection from any execution policy you may have in place.**

To check the current execution policy on your machine, type this command:

```
get-execution policy
```

Figure 10-3 shows the results on a machine with the default settings. Notice that the value of the `ExecutionPolicy` key is currently set to `Restricted`. When I try to execute a simple script, execution of the script does not take place and an error message is displayed.

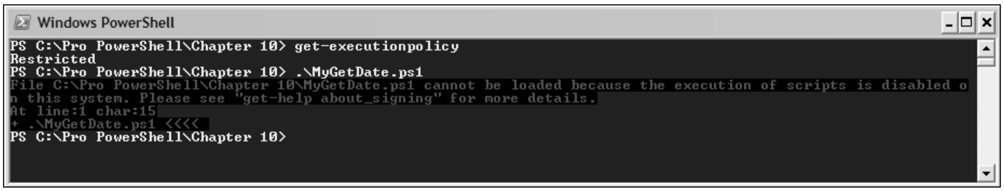


Figure 10-3

You can access the same information using the Registry Editor, as shown in Figure 10-4.

> **To change the setting for the Execution Policy by using the Registry Editor, you need to have administrator privileges on the machine.**

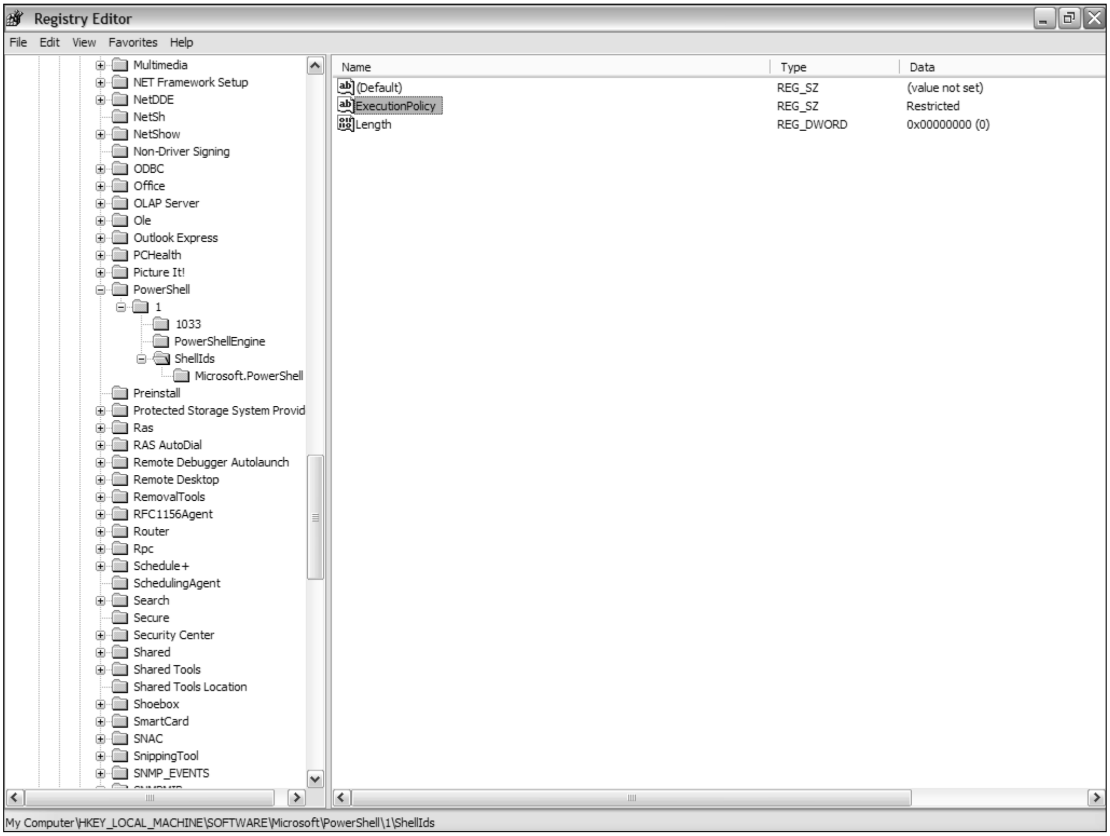


Figure 10-4

To modify the value for Execution Policy in the Registry Editor, right-click on Execution Policy, select Modify from the context menu then enter a valid, desired value in the dialog box that is displayed (see Figure 10-5).

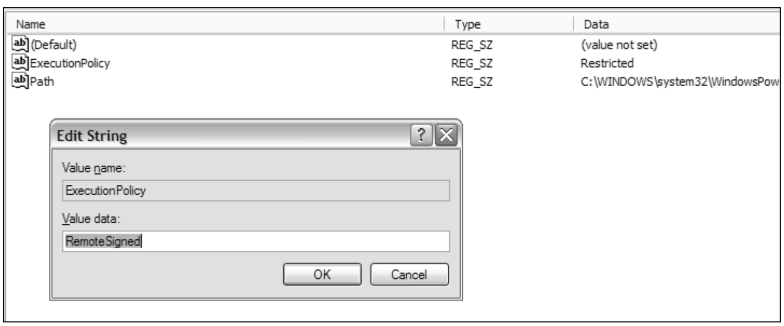


Figure 10-5

If you prefer, you can modify the value for `ExecutionPolicy` by using Windows PowerShell. Type the following command (assuming that your current location is `HKLM:\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell>`) to change the setting to `RemoteSigned`:

```
set-itemproperty -path . -name ExecutionPolicy -value RemoteSigned
```

Then check that you have successfully changed the value by using the following command:

```
get-itemproperty .
```

or:

```
get-executionpolicy
```

Figure 10-6 shows the value for `ExecutionPolicy` successfully changed to `RemoteSigned`. Windows PowerShell should recognize the change in execution policy immediately. If the `ExecutionPolicy` has not changed, then check the `set-itemproperty` statement for any errors.

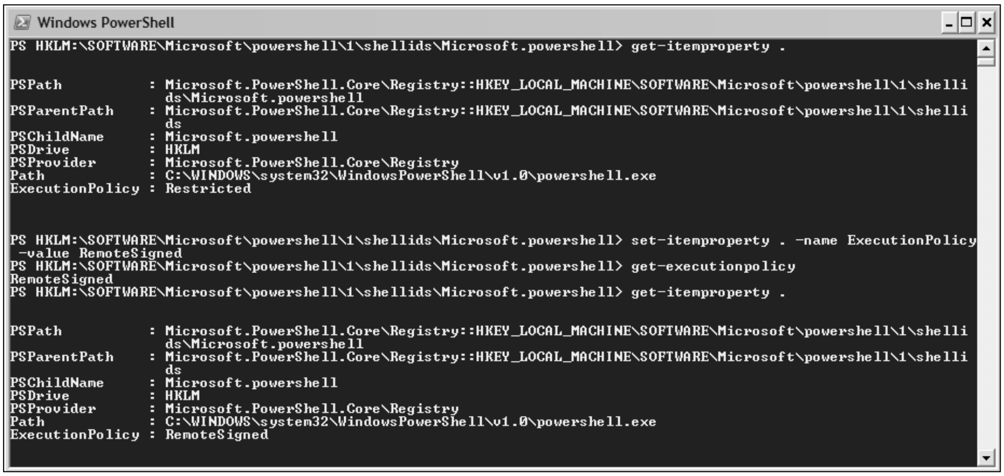


Figure 10-6

Once you have changed the execution policy to `RemoteSigned`, you should be able to run any scripts that you create locally. Here is a simple test script, `SimpleTest.ps1`, that you can use to confirm that you can run scripts.

```
$a = read-host "Enter a number to assign to the variable "'$a';
write-host '$a'"= $a";
```

The first line uses the `read-host` cmdlet to get some input from the user. The second line uses the `write-host` cmdlet to tell the user that the value entered was assigned to the variable `$a`. Up to this point, you have entered input directly on the command line, and the default formatter (covered in Chapter 7) has displayed the output. When running scripts, the `read-host` and `write-host` cmdlets are useful to capture input from the user and to display desired output. I describe both cmdlets later in this section.

To confirm that you can now execute Windows PowerShell scripts, type the following command (it assumes that you have saved a Windows PowerShell script, `SimpleTest.ps1`, in the `C:\Pro Monad\Chapter 10` directory):

```
& "C:\Pro PowerShell\Chapter 10\SimpleTest.ps1
```

If your current working directory is the folder you saved the script in, you can run it using this command:

```
. \SimpleTest.ps1
```

or:

```
. \SimpleTest
```

Figure 10-7 shows the script `SimpleTest.ps1` executed using each of the preceding three options.

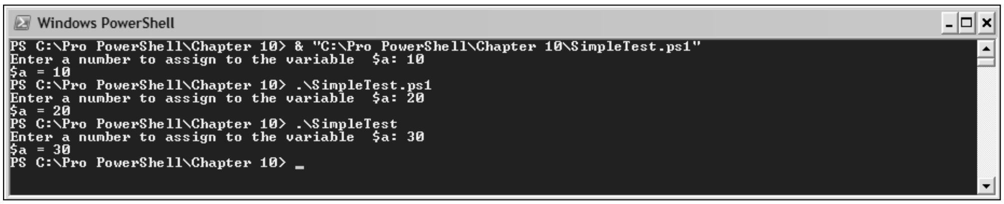


Figure 10-7

To view the Windows PowerShell Help file about permissions relating to signing scripts, use the following command:

```
help about_signing
```

Using the read-host Cmdlet

The `Read-Host` cmdlet reads a line of input from the command line. It supports the common parameters described in Chapter 6 and the following parameters:

- `Prompt` — A positional parameter in position 1, which takes as its value a string that will become the prompt to the user. If the prompt includes one or more space characters the value must be enclosed in paired quotation marks or paired apostrophes.
- `AsSecureString` — A boolean parameter that is optional. If it is `$true`, then the string input by a user is echoed as * characters.

The following script, `ReadHostTest.ps1`, demonstrates how the `read-host` cmdlet can be used.

```
$name = read-host "Enter your name ";
$password = read-host "Enter your password " -AsSecureString;
write-host "Your name is $name";
write-host "Your password is $password"
```

Two variables, `$name` and `$password`, are used to hold the values of user-entered information. The positional parameter used in the preceding code is the `prompt` parameter. Notice in Figure 10-8 that a colon character is automatically added to the prompt text that you provide in your script. The `$password` variable holds a password. Notice too that the `-AsecureString` parameter is used when the password is entered and that in Figure 10-8 the password is echoed to the screen as asterisks. When you try to display the value of `$password`, Windows PowerShell simply displays `System.Security.SecureString`.

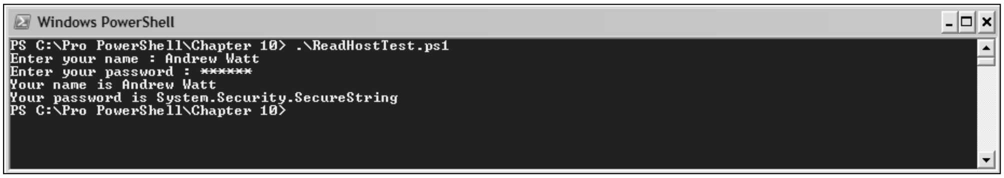


Figure 10-8

Notice that the use of the `SecureString` parameter with the `read-host` cmdlet means that the corresponding variable `$password` cannot be displayed in plain text using the `write-host` cmdlet (described next). Instead, the user only sees that it is a `System.Security.SecureString` object.

It's important that you know how secure the value of `$password` is. As written, `ReadHostTest.ps1` uses variables with script scope. So, once the script has ended `$password` can't be accessed from the PowerShell command line, as you can see in Figure 10-9. If you type

```
$password
```

there is no output, because outside the scope of the script no `$password` variable exists.

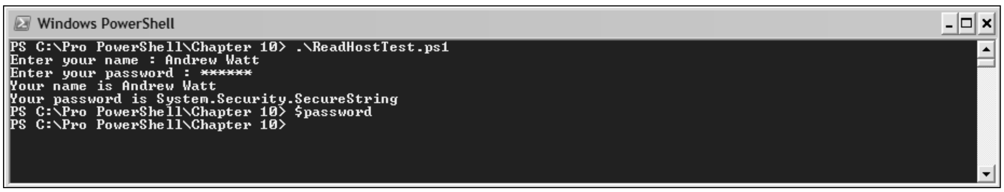


Figure 10-9

If you made the `$name` and `$password` variables global by modifying the script, as in `ReadHostTest2.ps1`, the `$password` variable continues to exist, but you can't get its value just by typing

```
$password
```

at the command line, as you can see in Figure 10-10. PowerShell simply tells you that `$password` is a `System.Security.SecureString` object.

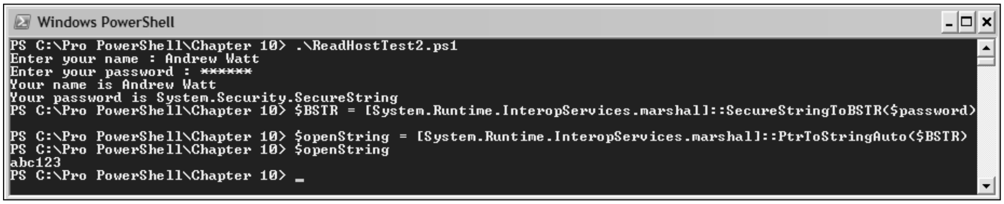


Figure 10-10

The following code reveals the value of the `System.Security.SecureString` object, which is `$password`.

```
$BSTR = [System.Runtime.InteropServices.marshal]::SecureStringToBSTR($password)
$openString = [System.Runtime.InteropServices.marshal]::PtrToStringAuto($BSTR)
$openString
```

The `SecureStringToBSTR()` method allocates a `BSTR` (basic string) and copies the content of a secure string into it. The `PtrToStringAuto()` method allocates a managed string, `$openString`, and copies the value of the BSTR into it. You can then display the value of `$openString` as shown in Figure 10-10. Of course, the password displayed is one that is inappropriate for use in real life.

Using the write-host Cmdlet

The `write-host` cmdlet displays specified objects to the console. When you use many cmdlets, you don't need to use the `write-host` cmdlet, since the objects passed along a pipeline are displayed by default by the default formatter (described in Chapter 6). On other occasions, for instance in the first example in the preceding `read-host` section, if you want to see the value of a variable, you can use the `write-host` cmdlet to display it, but it's not necessary to write

```
write-host $name
```

since the command

```
$name
```

will output the value of the `$name` variable to the console. The `write-host` cmdlet becomes more useful when you want to customize the display in some way. For example, you can specify the background color or the foreground color of the value(s) to be displayed.

In addition to the common parameters the `write-host` cmdlet supports the following parameters:

- `Object` — A positional parameter in position 1. Specifies the object (or objects) that are to be written to the console.
- `NoNewLine` — A boolean parameter. If present, after a line is written to the console, it is not followed by a newline.
- `Separator` — A string to be output to the console when multiple objects are processed by the `write-host` cmdlet. The value of the string is used as the separator between the values to be displayed.

- `BackgroundColor` — Specifies the background color.
- `ForegroundColor` — Specifies the foreground color.

You use the write-host cmdlet when you want to output information that is not being processed along a pipeline. In such scenarios, no output is displayed by default. For example, in the following code, `WriteHostTest.ps1`, the two values read in from the command line would simply exist as the variables `$name` and `$password` and not be displayed (since the two `write-host` statements are commented out). In some settings, that may be what you want. However, if objects are being passed along the pipeline (such as in the `get-process` statement), the objects are passed to the default formatter (in the absence of, say, a `format-list` or `format-table` statement) and then displayed.

```
$name = read-host "Enter your name ";
$password = read-host "Enter your password " -AsSecureString;
#write-host "Your name is $name";
#write-host "Your password is $password";

get-process svc*
```

Run the code with this command, assuming that the script file is in the current directory:

```
.\WriteHostTest.ps1
```

Figure 10-11 shows the result.

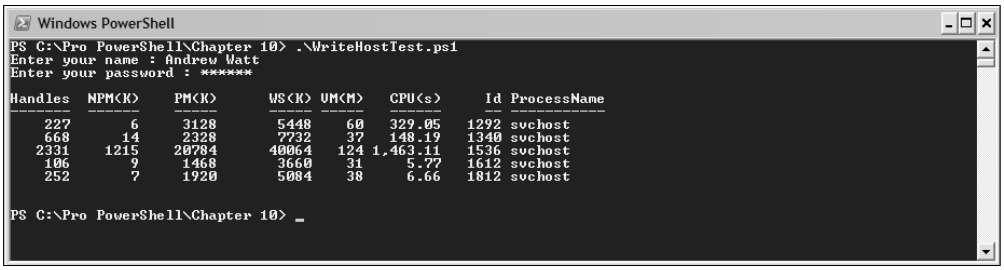


Figure 10-11

If you want to display the values of `$name` and `$password` later, uncomment the lines with the `write-host` cmdlet. If you want the value of `$password` to be visible in clear text, use the technique I showed you in the preceding section on the `read-host` cmdlet.

The `-foregroundColor` and `-backgroundColor` parameters of the `write-host` cmdlet allow you to alter the color of the text or the background when writing to the host, using the `write-host` cmdlet. The parameters support the named colors enumerated in the following table.

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

One use of the `write-host` cmdlet is to highlight prompts to the user. For example, instead of using `read-host` alone to read in data from the user (with the prompt being supplied as the value of the `-prompt` parameter of `read-host`), you can use the `write-host` cmdlet to display a more visible prompt to the user, then use the `read-host` cmdlet when capturing the user's input in a variable. The following example shows you how.

Save the following code as `WriteHostTest2.ps1`. Notice that the `-NoNewLine` parameter is used to keep the cursor at the end of the line where the text specified as the value of the `write-host` cmdlet is displayed. Notice too that the `-foregroundcolor` and `-backgroundcolor` parameters are used to highlight the prompt to the user.

```
write-host "Enter your name: " -NoNewLine -foregroundcolor black

-backgroundcolor white;
$name = read-host;
write-host "Enter your password: " -NoNewLine -foregroundcolor black

-backgroundcolor white;
$password = read-host -AsSecureString;
```

Figure 10-12 shows the results of executing the code.

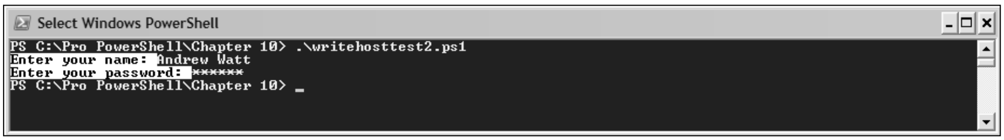


Figure 10-12

You can, of course, abbreviate the names of the `-foregroundcolor` and `-backgroundcolor` parameters as follows:

```
write-host "Enter your name: " -NoNewLine -fo black
code w/screen:$name = read-host;
write-host "Enter your password: " -NoNewLine -fo black
code last w/screen:$password = read-host -AsSecureString;
```

You may want to use the `write-host` cmdlet to display warning and informational messages in particular color combinations. The following code, `WriteHostColorTest.ps1`, displays a warning in red text on white and information in black text on white.

```
$LongEnough = $false
while(!$LongEnough)
{
$name = read-host "Enter your name ";
$password = read-host "Enter your password " -SecureString;
If ($password.length -ge 8)
 {
 $LongEnough = $true
 }
If ($LongEnough -eq $false)
 {
```

```
 write-host "Your password is not long enough!!"-backgroundcolor
white -foregroundcolor red
 write-host "Ensure your password is at least 8
characters." -backgroundcolor white -foregroundcolor red
 }
}

write-host "Your name is $name"
write-host "Your password is $password"
write-host "Thank you for providing your user credentials."
GX:To run the code, type:
```

```
.\WriteHostColorTest.ps1
```

Figure 10-13 shows the results after once entering a password with six characters, then entering a password with the required eight characters.

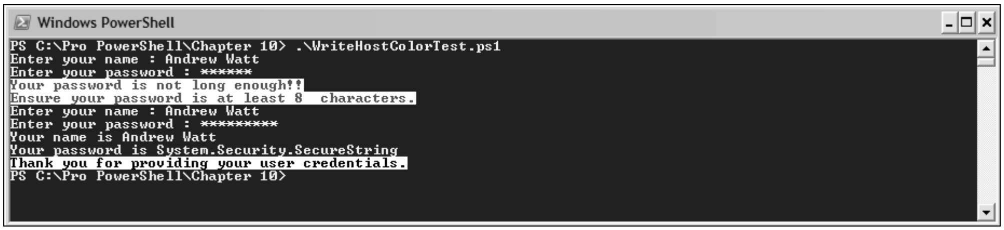


Figure 10-13

The variable `$LongEnough` is set initially to False. A `while` loop tests whether or not the `length` property of the `$password` variable is at least eight characters. If it is, then the value of `$LongEnough` is set to True. If not, a warning is displayed (in red text on a white background) and the user is prompted to reenter the user name and password. The following `write-host` statement specifies the red text on white:

```
write-host "Your password is not long enough!!"-backgroundcolor
white -foregroundcolor red
write-host "Ensure your password is at least 8
characters."-backgroundcolor white -foregroundcolor red
```

Once a password of the specified length has been entered, the `while` loop exits and an informational message, specified in the following code, is displayed in black text on white:

```
write-host "Thank you for providing your user credentials."
Heading 1:Windows PowerShell Operators
```

In this section, I am going to take a look at the range of operators that Windows PowerShell supports. The behavior and syntax of many operators is likely to be familiar to you. Windows PowerShell supports the following types of operators:

- **Arithmetic** — Use to calculate values
- **Assignment** — Use to assign one or more values to a variable
- **Comparison** — Use to compare values and perform conditional tests

- **Logical** — Use in statements containing more than one conditional test, to specify how those tests are to be applied
- **Unary** — Use to increment or decrement variables or object properties
- **Special** — Use to, for example, run commands or specify a value's datatype

I describe each of the operators later in this section and show you how they can be used.

The Arithmetic Operators

Windows PowerShell supports many arithmetic operators that are likely familiar to you from other languages. The supported operators are:

Operator	Use
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

Figure 10-14 shows very simple examples using each of the five arithmetic operators in the preceding list. The result of each calculation is displayed on the screen following the use of each operator.

Figure 10-14

You can, of course, use the arithmetic operators to manipulate numeric values held as variables. The following example shows the addition of two variables. Type these commands:

```
$a = 10
$b = 5
$total  = $a + $b
$total
```

Figure 10-15 shows the results. The final line causes the value of `$total (15)` to be displayed on the console.

Figure 10-15

To view the help file about the arithmetic operators, type the following command at the prompt:

```
help about_arithmetic_operator
```

Operator Precedence

When evaluating arithmetic operators, Windows PowerShell evaluates expressions based on the following order of precedence:

1. – (indicating a negative number)
2. `*`, `/`, and `%`
3. `+`, – (indicating subtraction)

For example,

```
6 + 4 / 2
```

is the same as

```
6 + (4 / 2)
```

since the `/` operator has the higher precedence and is evaluated first.

Figure 10-16 shows simple examples that demonstrate some of the operator precedence just listed. Notice that

```
10 + -2 / 4
```

is equivalent to

```
10 + (-2 / 4)
```

since the `/` operator has greater precedence than the `+` operator.

Figure 10-16

To view the help file about operator precedence, type the following command at the prompt:

```
help about_arithmetic_operators
```

The Assignment Operators

Windows PowerShell supports several assignment operators, which are summarized in the following table.

Operator	Meaning
=	Assigns a value to a variable
+=	Adds the value of the right side of the assignment to the existing value of the left side and assigns the result to the variable on the left side.
-=	Subtracts the value of the right side of the assignment from the existing value of the left side and assigns the result to the variable on the left side.
*=	Multiplies the value of the right side of the assignment and the existing value of the left side and assigns the result to the variable on the left side.
/=	Divides the value of the left side of the assignment into the existing value of the right side and assigns the result to the variable on the left side.
%=	Divides the value of the left side of the assignment into the existing value of the right side and assigns the remainder to the variable on the left side.

The simplest assignment operator in the Windows PowerShell language is the = sign. The command

```
$a = 5
```

assigns the numeric value 5 to the variable `$a`.

To add `3` to `$a` and assign the result to `$a`, type this command:

```
$a += 3
```

It is equivalent to:

```
$a = $a + 3
```

To subtract `4` from `$a` and assign the result to `$a`, type this command:

```
$a -= 4
```

It is equivalent to:

```
$a = $a - 4
```

To multiply `$a` by `5` and assign the result to `$a`, type this command:

```
$a *= 5
```

It is equivalent to:

```
$a = $a * 5
```

To divide `$a` by `4` and assign the result to `$a`, type this command:

```
$a /= 4
```

To find the remainder from dividing `$a` by `2` and assign the remainder to `$a`, type this command:

```
$a %= 2
```

Figure 10-17 shows the results of entering the previous commands, then displaying the current value of the variable `$a`.

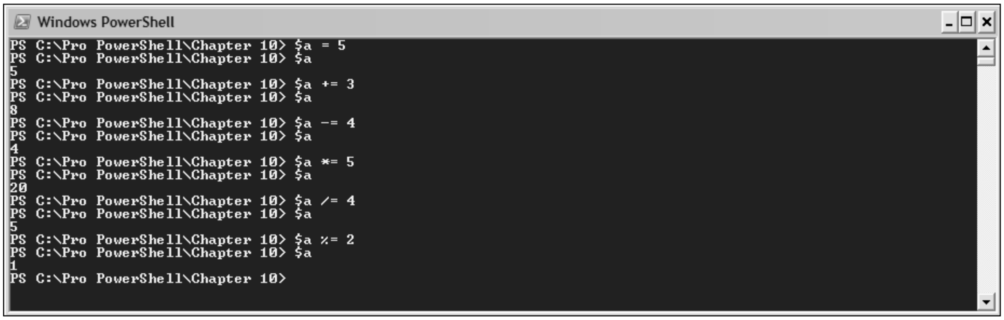


Figure 10-17

The preceding assignment operators are used with numeric values. However, you can also use assignment operators, where appropriate, with string values. For example, to assign the string value `Hello` to the variable `$myString`, type this command:

```
$myString = "Hello"
```

When assigning string values, use paired quotation marks or paired apostrophes to enclose the string value.

You can assign multiple values to multiple variables in a single assignment statement. The following statement assigns `1` to `$a`, `2` to `$b` and `3` to `$c`. The resulting values are displayed in Figure 10-18.

```
$a, $b, $c = 1, 2, 3
```

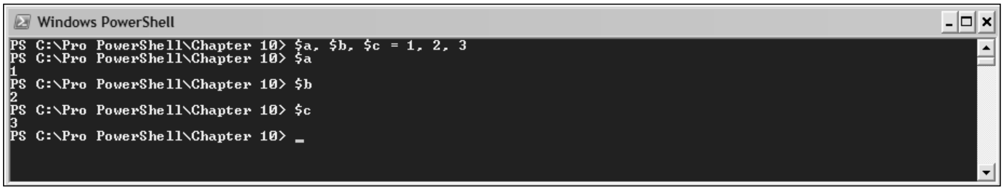


Figure 10-18

The assignment operator, =, can also be used to assign multiple values to a numeric array or a string array. I discuss arrays in more detail in Chapter 11. For example, the following command:

```
$myNumericArray = 1, 2, 3, 4
```

creates an array containing four elements each containing a numeric value. Similarly, the following command:

```
$myStringArray = "once", "twice", "thrice", "and more"
```

creates an array with four elements each of which contains a string value. Figure 10-19 shows the results.

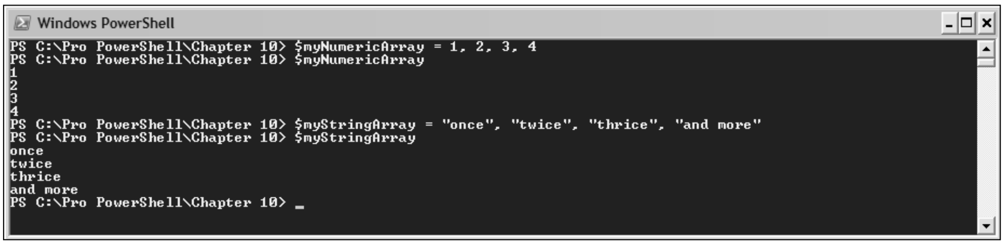


Figure 10-19

You can also use the = assignment operator to assign values to an associative array. An associative array is a data structure for storing collections of keys and values. Associative arrays are discussed in Chapter 11.

To view the help file about the assignment operators, type the following command at the prompt:

```
help about_assignment_operator
```

The Comparison Operators

Windows PowerShell supports several comparison operators. A comparison evaluates a conditional expression to the values of either `$true` or `$false`. Comparison operators enable you to perform some test (one variable greater than or equal to another, for example) and to use the result to determine whether or not a particular statement block is to be executed. I describe that use of comparison operators in more detail in Chapter 11.

Several operators are available to compare numeric and string values. When used to perform comparisons on string values the comparison using the operators in the following table is case-insensitive.

Operator	Description
`-eq`	Tests for equality.
`-ne`	Tests for inequality.
`-gt`	Tests whether the value on the left is greater than the value on the right.
`-ge`	Tests whether the value on the left is greater than or equal to the value on the right.
`-lt`	Tests whether the value on the left is less than the value on the right.
`-le`	Tests whether the value on the left is less than or equal to the value on the right.
`-like`	Tests, using wildcards, whether two values match. The wildcard(s) go on the right side.
`-notlike`	Tests, using wildcards, whether two values fail to match. The wildcard(s) go on the right side.
`-match`	Tests, using regular expressions, whether two values match. The regular expression goes on the right side.
`-notmatch`	Tests, using regular expressions, whether two values fail to match. The regular expression goes on the right side.

The following operators are constructed by adding a "c" to the operator name. Each test is case-sensitive.

Operator	Description
`-ceq`	Tests for case-sensitive equality.
`-cne`	Tests for case-sensitive inequality.
`-cgt`	Tests whether the value on the left is greater than the value on the right. Case-sensitive comparison.
`-cge`	Tests whether the value on the left is greater than or equal to the value on the right. Case-sensitive comparison.
`-clt`	Tests whether the value on the left is less than the value on the right. Case-sensitive comparison.
`-cle`	Tests whether the value on the left is less than or equal to the value on the right. Case-sensitive comparison.
`-clike`	Tests, using wildcards, whether two values match. The wildcard(s) go on the right side. Case-sensitive comparison.

Table continued on following page

Operator	Description
`-cnotlike`	Tests, using wildcards, whether two values fail to match. The wildcard(s) go on the right side. Case-sensitive comparison.
`-cmatch`	Tests, using regular expressions, whether two values match. The regular expression goes on the right side. Case-sensitive matching.
`-cnotmatch`	Tests, using regular expressions, whether two values fail to match. The regular expression goes on the right side. Case-sensitive matching.

Although the comparison operators such as `eq` and `gt` are used case-insensitively, Windows PowerShell also provides explicitly case-insensitive comparison operators, which are described in the following table.

Operator	Description
`-ieq`	Tests for case-insensitive equality.
`-ine`	Tests for case-insensitive inequality.
`-igt`	Tests whether the value on the left is greater than the value on the right. Case-insensitive comparison.
`-ige`	Tests whether the value on the left is greater than or equal to the value on the right. Case-insensitive comparison.
`-ilt`	Tests whether the value on the left is less than the value on the right. Case-insensitive comparison.
`-ile`	Tests whether the value on the left is less than or equal to the value on the right. Case-insensitive comparison.
`-ilike`	Tests, using wildcards, whether two values match. The wildcard(s) go on the right side. Case-insensitive comparison.
`-inotlike`	Tests, using wildcards, whether two values fail to match. The wildcard(s) go on the right side. Case-insensitive comparison.
`-imatch`	Tests, using regular expressions, whether two values match. The regular expression goes on the right side. Case-insensitive matching.
`-inotmatch`	Tests, using regular expressions, whether two values fail to match. The regular expression goes on the right side. Case-insensitive matching.

The `-replace` operator is described in Chapter 12.

To view the help file about the comparison operators, type the following command at the prompt:

```
help about_comparison_operator
```

The Logical Operators

Windows PowerShell supports four logical operators, which are used to combine tests using the comparison operators described in the preceding section. They are described in the following table.

Operator	Meaning
`-and`	Is true if both comparisons are true and only then.
`-or`	Is true if one or both comparisons is true.
`-not`	Negation.
`!`	Negation. Synonym for `-not`.

To test whether 3 is greater than 4, type this command:

```
(3 -gt 4)
```

Not surprisingly, it returns `false`. To test whether 5 is greater than or equal to 3, type this command:

```
(5 -ge 3)
```

This returns `true`.

Using the `-and` logical operator, you can test whether both comparisons are true with the following command:

```
(3 -gt 4) -and (5 -ge 3)
```

This returns `false` since the first test returns `false`. Because it is then impossible for both comparisons to be true, you know that the overall test is false. However, if you use the `-or` logical operator:

```
(3 -gt 4) -or (5 -ge 3)
```

the first test returns `false` and the right test returns `true`. Since only one part of the test needs to be true for the overall test to succeed, when you use the `-or` operator the overall test returns `true`.

The results from these examples are shown in Figure 10-20.

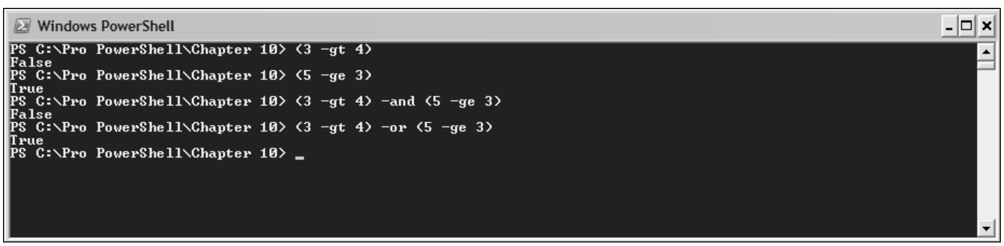


Figure 10-20

To view the help file about the logical operators, type the following command at the prompt:

```
help about_logical_operator
```

The Unary Operators

Windows PowerShell supports the unary operators listing in the following table.

Operator	Meaning
+	Signifies explicitly that a number is a positive number
-	Signifies that a number is a negative number
++	Increments a value or variable
--	Decrements a value or variable

The decrement and increment operators work similarly to the equivalent operators in many other languages.

To assign `5` to `$a` then increment it, type these commands:

```
$a = 5
$a++
```

The value of `$a` is now `6`.

To assign `10` to `$b` and then decrement it, type these commands:

```
$b = 10
$b--
```

The results are shown in Figure 10-21.

Figure 10-21

In some settings, you need to be careful whether the increment or decrement operators come before or after a variable name.

Set `$a` to `10` using the following command:

```
$a = 10
```

then type the following command:

```
$b = $a++
```

That assigns the value of `$a` to `$b` and after that assignment has taken place, it then increments `$a`. As you can see in Figure 10-22, the value of `$b` is `10` and the value of `$a` is `11`.

Figure 10-22

However, if you assign the value `10` to `$a`, then type the following command:

```
$c = ++$a
```

the value of `$a` is incremented *before* the assignment. So, `$a` is `11` when the assignment takes place. Therefore, as you can see in the lower part of Figure 10-22, the value of both `$a` and `$c` is `11`.

Using the set-variable and Related Cmdlets

Windows PowerShell supports several cmdlets that allow you to assign a value to a variable or otherwise manipulate variables. They are:

- `set-variable`
- `new-variable`
- `get-variable`
- `clear-variable`
- `remove-variable`

I describe each of these cmdlets in the following sections.

The set-variable Cmdlet

The `set-variable` cmdlet provides an alternative to the assignment operator described earlier in this chapter, to allow you to assign a value or values to a variable. One use is to allow variable assignment in a pipeline.

In addition to the common parameters, the `set-variable` cmdlet supports the following parameters:

- `Name` — Specifies the name of the variable being set. It is a positional parameter in position 1. A value must be specified.
- `Include` — Specifies only those items upon which the cmdlet will act.
- `Exclude` — Specifies those items upon which the cmdlet will not act.
- `Scope` — The scope where the variable is to be created — which can be a named scope ("global", "local", or "script") or a number relative to the current scope (0 through the number of scopes, where 0 is the current scope and 1 is its parent).
- `Value` — Specifies the value assigned to the variable.
- `Description` — Specifies a user-defined description of the variable.
- `Option` — Allowed values are `None`, `ReadOnly`, `Constant`, `Private`, `AllScope`.
- `Force` — Specifies that every effort be made to set the variable.
- `Whatif` — A boolean value that specifies that no action should be taken, but the user should be shown what would have happened if the cmdlet had executed.
- `Confirm` — A boolean value that specifies that the user be asked to confirm the intended action before it is carried out.
- `Passthru` — Specifies that the object(s) created are passed to the next step in the pipeline.

The following shows an example of using the `set-variable` cmdlet. Type the following command to assign the value of `20` to `$a`:

```
set-variable -name a -value 20 |
format-list
```

The `format-list` statement displays nothing, as you can see in top lines shown in Figure 10-23. This is because there is no `-passthru` parameter specified. If you add a `-passthru` parameter, as shown here:

```
set-variable -name a -value 20 -passthru -description "A demo variable" | format-
list
```

then information about the variable can be displayed using the `format-list` statement (shown in the bottom portion of Figure 10-23).

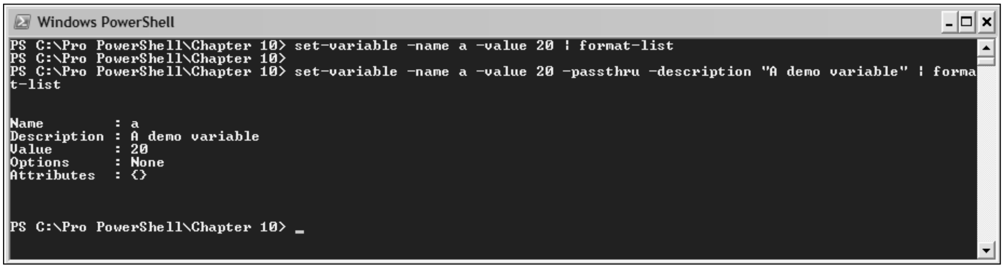


Figure 10-23

The new-variable Cmdlet

The `new-variable` cmdlet creates a new variable.

In addition to the common parameters, the `new-variable` cmdlet supports the following parameters:

- `Name` — Specifies the name of the variable to be created. A positional parameter in position 1.
- `Value` — Specifies a value for the variable. A positional parameter in position 2.
- `Description` — Specifies a description for the variable.
- `Option` — Specifies options relating to the variable. Permitted values are `None`, `ReadOnly`, `Constant`, `Private`, `AllScope`.
- `Force` — Specifies that every effort will be made to create the variable.
- `Passthru` — Specifies that the object(s) created are passed to the next step in a pipeline.
- `Scope` — Specifies the scope of the variable.
- `Whatif` — A boolean value that specifies that no action is taken but the user is shown what would have happened if the cmdlet had executed.
- `Confirm` — A boolean value that specifies that the user is asked to confirm the intended action before it is carried out.

The `New-Variable` cmdlet does not take `include` or `exclude` parameters, unlike the `set-variable` cmdlet, which supports those parameters.

The following command creates a new variable — `$myNewVariable` — but does not assign a value to it. Since the `passthru` parameter is present, the `format-list` statement allows you to see, in Figure 10-24, that the variable exists but has no value set.

```
new-variable -name myNewVariable -passthru |
format-list
```

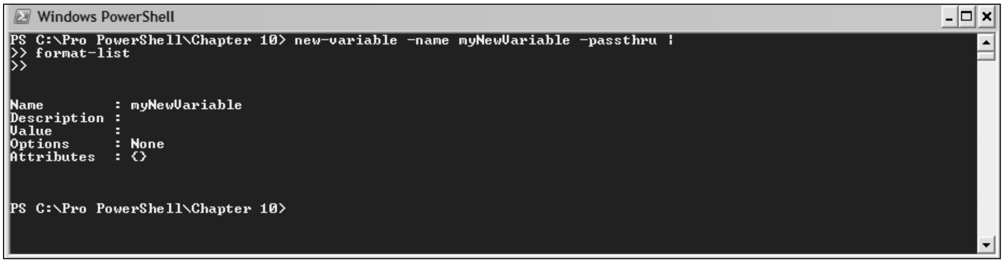


Figure 10-24

The get-variable Cmdlet

The `get-variable` cmdlet allows you to retrieve a Windows PowerShell variable. The `get-variable` cmdlet supports the following parameters in addition to the common parameters:

- `Name` — Specifies the name of the variable(s). Accepts wildcard characters.
- `ValueOnly` — A boolean value. If present, then only the value of the variable (not the object) is passed along the pipeline.
- `Include` — If present, specifies which variables to include. Qualifies the value of the `-name` parameter.
- `Exclude` — If present, specifies which variables to exclude. Qualifies the value of the `-name` parameter.
- `Scope` — Specifies the scope of the variable(s).

The following example shows the creation of four variables, `$a1`, `$a2`, `$a3`, and `$a4`. The `get-variable` cmdlet is used to retrieve and display information about the variables `$a1`, `$a2`, and `$a4`.

Type these commands to create the variables:

```
$a1 = 10
$a2 = 20
$a3 = 30
$a4 = 40
```

Type this command to retrieve the previously mentioned variables:

```
get-variable -name a* -include a[124] |
format-list
```

As you can see in Figure 10-25, information on the three desired variables is displayed. The value of the `-include` parameter is a regular expression pattern. The character class `[124]` matches a single character, which is contained in the class. Thus the variables `$a1`, `$a2`, and `$a4` match, but `$a3` does not match, since 3 isn't contained in the character class.

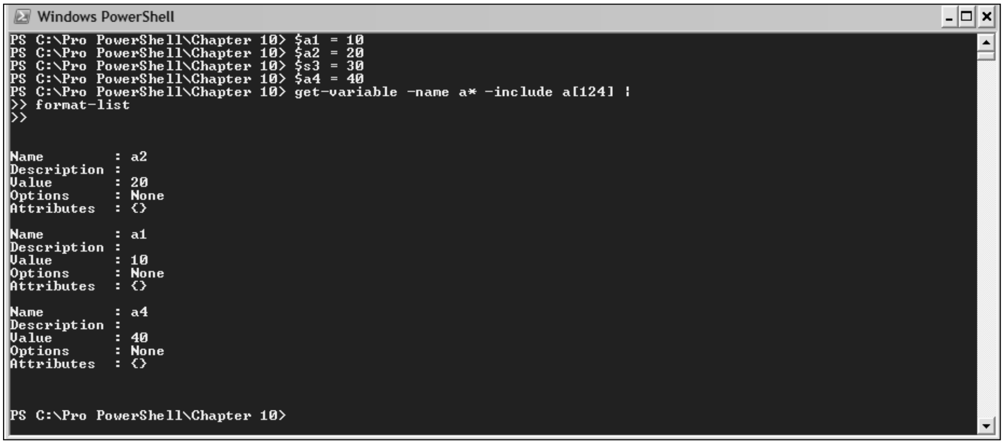


Figure 10-25

Notice that the variables are not, by default, ordered by name. If you wanted to sort them by name, you would need to add a pipeline step, using the `sort-object` cmdlet.

```
get-variable -name a* -include a[124] |
sort-object Name |
format-list
```

The clear-variable Cmdlet

The `clear-variable` cmdlet clears the value(s) of one or more variables.

In addition to the common parameters, the `clear-variable` cmdlet supports the following parameters:

- `Name` — The name of the variable(s) whose value(s) are to be cleared
- `Include` — A filter that includes a subset of the name(s) specified by the `Name` parameter
- `Exclude` — A filter that excludes a subset of the name(s) specified by the `Name` parameter
- `Force` — Specifies that every effort will be made to create the variable
- `Scope` — Specifies the scope of the variable(s)
- `Whatif` — A boolean value that specifies that no action should be taken but the user should be shown what would have happened if the cmdlet had executed
- `Confirm` — A boolean value that specifies that the user be asked to confirm the intended action before it is carried out

The following example demonstrates clearing the value of specified variables. You will clear the values of the `$a1`, `$a2`, `$a3`, and `$a4` variables created in the preceding section. First, show that the four variables exist and have a value set using this command:

```
get-variable -name a* -include a[1234] |
format-list name, value
```

Next clear the value of three of those variables using this command. The value of `$a3` is not cleared.

```
clear-variable -name a* -include a[124]
```

Then check to see that the values of those variables have been cleared using this command:

```
get-variable -name a* -include a[1234] | format-list name, value
```

Figure 10-26 shows the results. As you can see, the value of each of the three variables has been cleared.

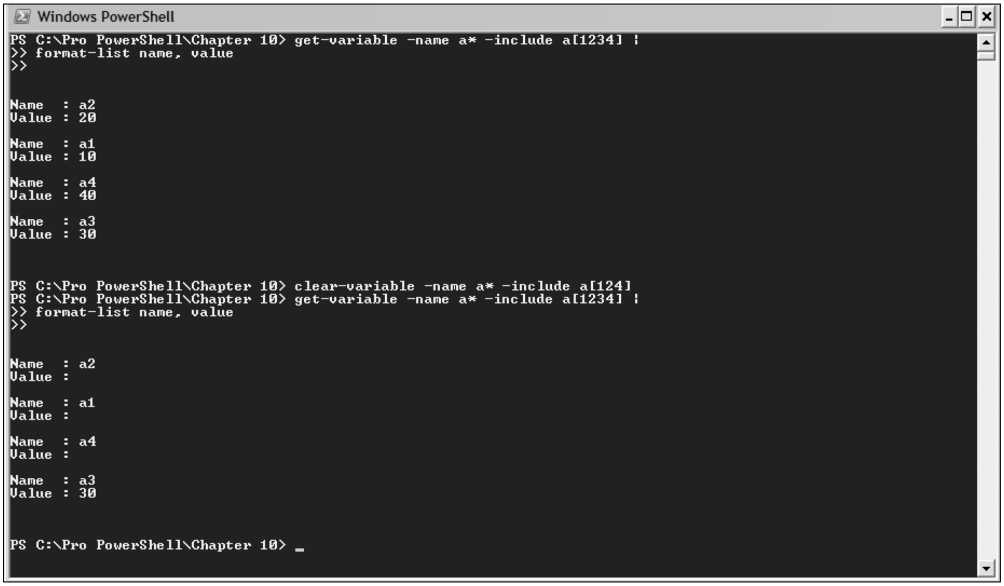


Figure 10-26

The remove-variable Cmdlet

The `remove-variable` cmdlet removes one or more existing variables.

The `remove-variable` cmdlet supports the following parameters in addition to the common parameters:

- `Name` — The name of the variable(s) to be removed
- `Include` — Specifies a subset of the variables specified by the value of the `Name` parameter that are to be removed
- `Exclude` — Specifies a subset of the variables specified by the value of the `Name` parameter that are not to be removed
- `Force` — Specifies that every effort is to be made to remove variables

- `Scope` — Specifies the scope of the variable(s)
- `Whatif` — A boolean value that specifies that no action should be taken but the user is to be shown what would have happened if the cmdlet had executed
- `Confirm` — A boolean value that specifies that the user be asked to confirm the intended action before it is carried out

In the following example, you delete the variables `$a1`, `$a2`, and `$a3`. Should you be unsure of the effect of using a `remove-variable` command, you can use the `whatif` parameter. As in the preceding section, I use four variables, `$a1`, `$a2`, `$a3`, and `$a4`, for the example. To explore removing the three desired variables with the protection of the `whatif` parameter, type this command:

```
remove-variable -name -include a[123] -whatif
```

As you can see in Figure 10-27 there are three variables that would have been removed if the `whatif` parameter hadn't been used. Since those are the variables you desire to delete, remove the `whatif` parameter from the command:

```
remove-variable -name -include a[123]
```

Without further warning, the variables are removed, as you can confirm by typing:

```
get-variable -name a* -include a[1-4]
```

When you execute the following command, you can confirm that only `$a4` still exists, as shown in Figure 10-27.

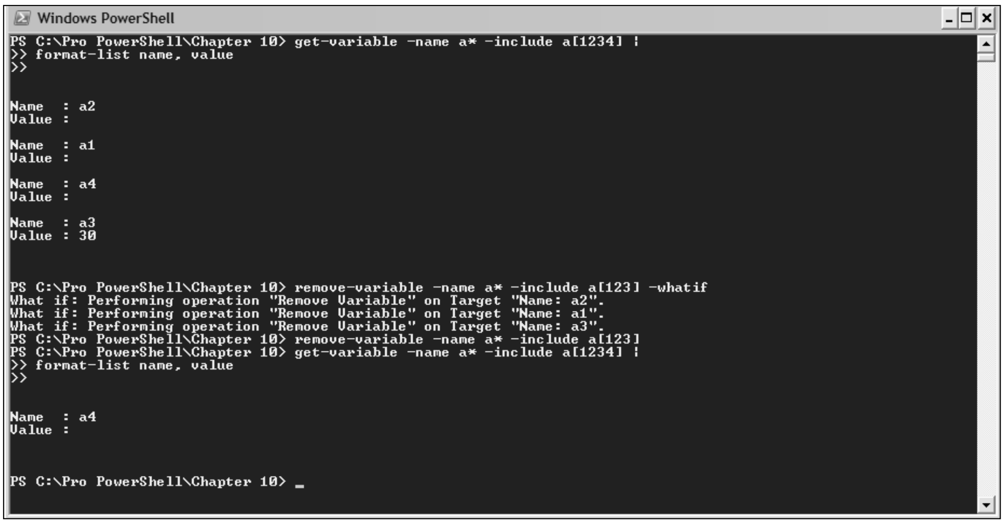


Figure 10-27

Summary

The default install of Windows PowerShell prevents you executing PowerShell scripts and configuration files on PowerShell startup. The `get-executionpolicy` cmdlet allows you to find out the current setting of the Windows PowerShell execution policy.

To modify the current execution policy, you can use the `set-executionpolicy` cmdlet, if you have administrator privileges. I showed you alternative techniques using Regedit or editing the registry from the Windows PowerShell command line.

The `read-host` cmdlet allows you to accept user input. The `write-host` cmdlet allows you to customize the display of information in the PowerShell console.

I described the following types of operators that Windows PowerShell supports:

- **Arithmetic** — Use to calculate values
- **Assignment** — Use to assign one or more values to a variable
- **Comparison** — Use to compare values and perform conditional tests
- **Logical** — Use in statements containing more than one conditional test, to specify how those tests are to be applied
- **Unary** — Use to increment or decrement variables or object properties
- **Special** — Use to, for example, run commands or specify a value's datatype

I introduced the following cmdlets that you can use to work with PowerShell variables:

- `set-variable`
- `new-variable`
- `get-variable`
- `clear-variable`
- `remove-variable`

Chapter 11 introduces several more features of the Windows PowerShell language.

11

Additional Windows PowerShell Language Constructs

In this chapter, I continue describing Windows PowerShell language constructs that are available to you for use in Windows PowerShell scripts. I cover the following topics:

- Arrays
- Associative arrays
- Conditional expressions
- Looping constructs
- The add-member cmdlet

Arrays

An array is a collection of data elements. In Windows PowerShell, an array can contain elements of any *type* supported by the .NET Framework. Array elements in the Windows PowerShell language are numbered from zero. The first element in the array is element 0, the second is element 1, and so on.

To create an array, assign multiple values to a variable. To create a simple array named `$myArray` containing three elements, type the following command:

```
$myArray = 1, 2, 3
```

You can display all elements in the array simply by typing

```
$myArray
```

at the Windows PowerShell prompt. If you want to display a selected element of the array, supply the number of the element in square brackets. For example, to display the first element of the `$myArray` variable, just type:

```
$myArray[0]
```

Figure 11-1 shows the execution of the preceding commands.

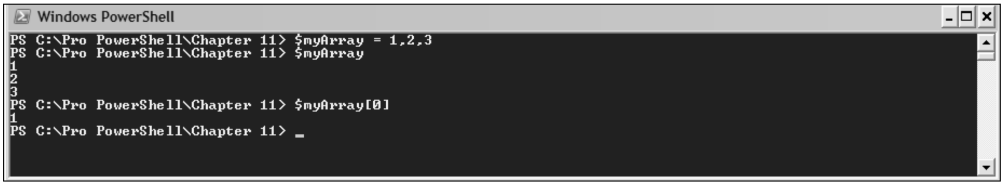


Figure 11-1

The names of arrays are case-insensitive, so typing:

```
$myarray
```

or:

```
$myArray
```

or any other variant of case will all display the values of the same array.

As I mentioned earlier, in Windows PowerShell, an array can contain various .NET types in individual elements of an array. To create an array with three .NET types, type the following at the command prompt:

```
$mixedArray = 1, "Hello", 2.55
```

To display the elements in the array, type:

```
$mixedArray
```

You can find the methods available on `$mixedArray` and on each of its elements by using the `get-member` cmdlet. The following command shows the methods available on the array:

```
$mixedArray | get-member
```

The members of some elements of the array, which is an array of objects, are shown in Figure 11-2. Notice that the members of the first element, `$mixedArray[0]`, which is a `System.Int32` value, are different from those of the second element, `$mixedArray[1]`, which is a `System.String`.

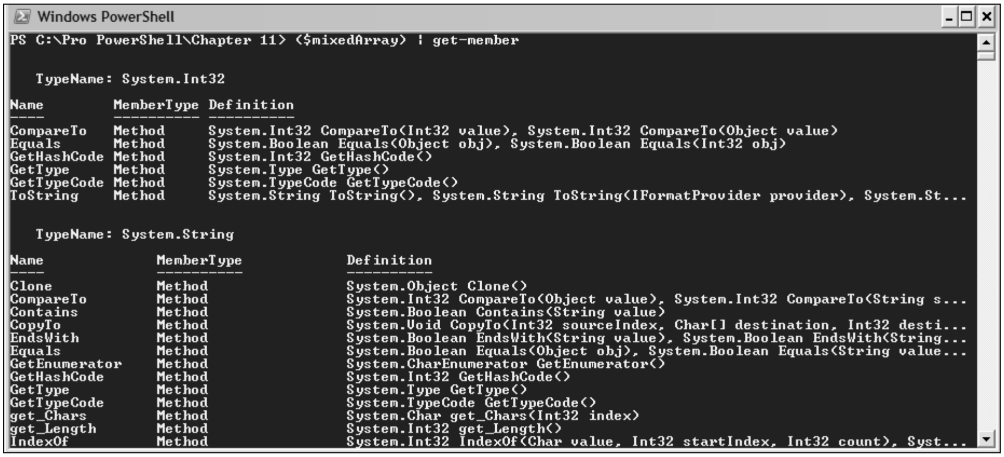


Figure 11-2

The `GetType()` method is available on the array and on each of its members. You can use the `GetType()` method on the array to see its type. To see the type of the `$mixedArray` array, type:

```
$mixedArray.GetType()
```

As you can see in Figure 11-3, the array contained in the variable `$mixedArray` is an array of objects, as indicated by the value of the `Name` property of a `System.Runtime` object. That explains why an array can hold values which are strings, integers, and so on.

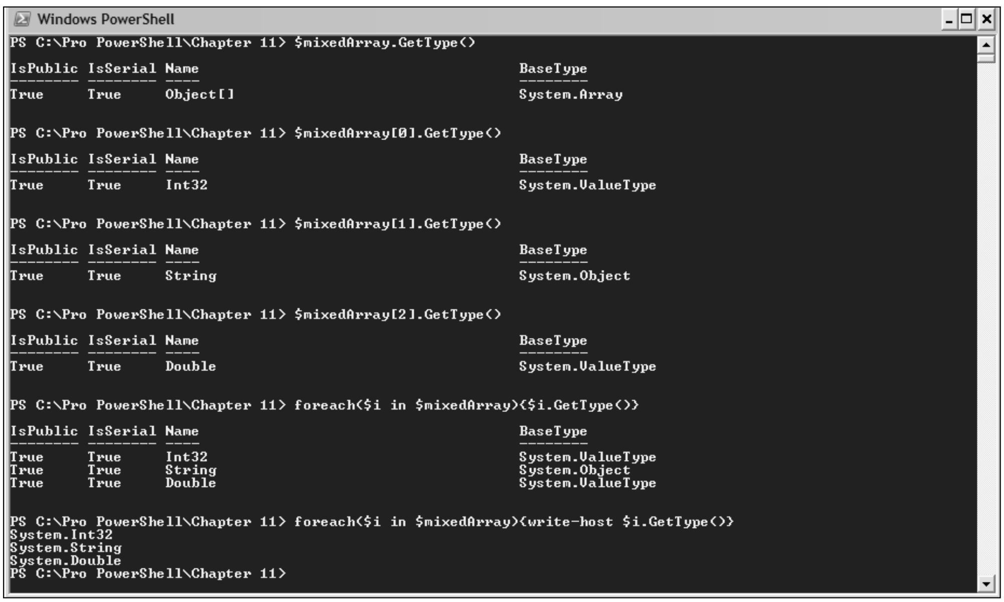


Figure 11-3

You can display the type of each element of the array by using the following individual commands:

```
$mixedArray[0].GetType()
$mixedArray[1].GetType()
$mixedArray[2].GetType()
```

with results also shown in Figure 11-3.

However, that approach is tedious even in small arrays. The PowerShell language has a construct, the `foreach` statement (that I describe in more detail later in this chapter), that allows you to iterate over each element of an array. To view type information on each element of the array, use this command:

```
foreach ($i in $mixedArray)
{
$i.GetType()
}
```

You can type the command over several lines, as in the preceding code, which aids clarity or type it on a single line as shown in the lower part of Figure 11-3. The `foreach` statement executes the code in the paired curly braces for each element in the array.

It's not immediately obvious that the type "String" is in the `System` namespace and is a `System.String`. To see the fully qualified name of a type use the `write-host` cmdlet inside the curly braces. If you type the following you can see the fully qualified type of each element in the array. The results are shown in the final part of Figure 11-3.

```
foreach($i in $mixedArray)
{
write-host $i.GetType()
}
```

The `foreach` statement, described later in this chapter, iterates through each element of the `$mixedArray` array and uses the `write-host` cmdlet to write the value returned by the `GetType()` method for each element of the array.

When creating an array, you can use the range operator to populate multiple elements in the array with successive values. For example, to assign the integers 8 to 12 to an array referenced by the `$useRange` array, type this command:

```
$useRange = 8..12
```

Alternatively, you can write it as:

```
$useRange = (8 .. 12)
```

You can display the values in the elements of the `$useRange` array, using the following command:

```
$useRange
```

The range operator, `..`, can only be used with integer values. You cannot, for example, use the range operator to populate an array with successive characters, as shown in Figure 11-4.

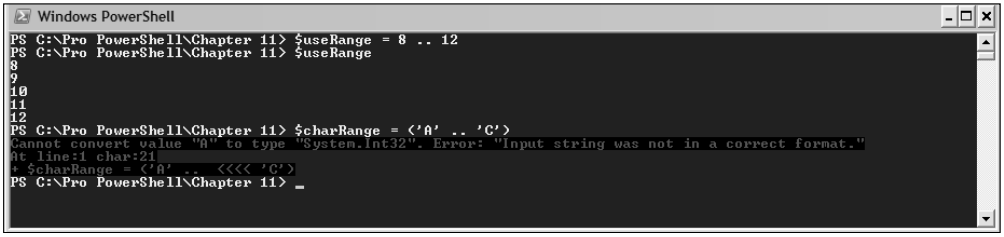


Figure 11-4

The two commands:

```
$myArray | get-member
```

and:

```
get-member -inputObject $myArray
```

do not display the same results. The first command displays the members of array elements in `$myArray`. The second command displays the member of the array itself.

Creating Typed Arrays

The arrays you have created so far are arrays of .NET objects. As you saw when you created `$mixedArray`, you can use several .NET types in a single array. PowerShell provides syntax to allow to you strictly type an array so that all elements must be of a specified .NET type.

To specify that an array consists of values that are of type `System.Int32`, use the following command:

```
[System.Int32[]]$integerArray = (1, 11, 99, 235)
```

Display the values in a typed array simply by typing the array name, as before:

```
$integerArray
```

Figure 11-5 shows the results.

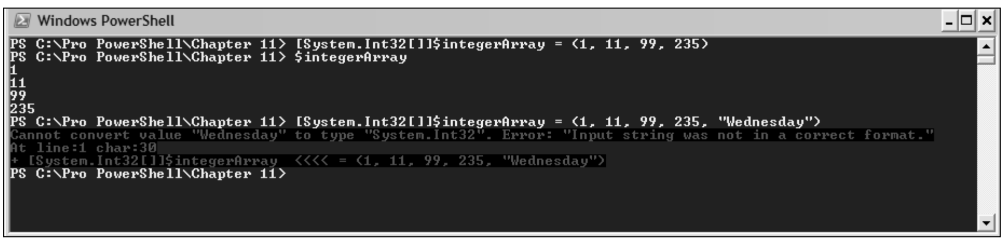


Figure 11-5

If you attempt to include a value that is not of the specified type, an error message is displayed, as shown in Figure 11-5.

```
[System.Int32[]]$integerArray = (1, 11, 99, 235, "Wednesday")
```

However, in some situations, PowerShell will automatically cast a value to the type specified for the array. For example, if you create an array of strings using the following command

```
[System.String[]]$stringArray = ("Hello", "world", "it's", "Wednesday", 55)
```

the final value is an integer, as written inside the parentheses. However, since 55 can be cast to a string, PowerShell treats the value as a string, as you can see in Figure 11-6 by typing the following command.

```
$stringArray[4].GetType()
```

Figure 11-6

When creating a typed array, you can create elements of any desired .NET 2.0 type. The following command creates an array of `System.ServiceProcess.ServiceController` objects.

```
[System.ServiceProcess.ServiceController[]]$services = get-service
```

You can confirm that each element is of the desired type by typing either of the following commands:

```
$services[0].GetType()
```

or:

```
write-host $services[0].GetType()
```

Figure 11-7 shows the results of executing the preceding commands.

Figure 11-7

Modifying the Structure of Arrays

Windows PowerShell supports several ways of modifying the structure of an existing array, which I demonstrate in the following examples.

To set the value of an array element to `null`, simply assign the `$null` variable (the Windows PowerShell way of expressing a null value) to the array element you want to change(s). You can see an example of this in the following command, which sets the value of the third element in an array to `null`.

```
$myArray[2] = $null
```

Notice in Figure 11-8 that when you display all elements of the array, nothing is displayed for the third element of the array. However the array element is still there. You can demonstrate that using this command:

```
$myArray.length
```

The `Length` property of `$myArray` reflects the length of the array (the number of elements it has) and remains three.

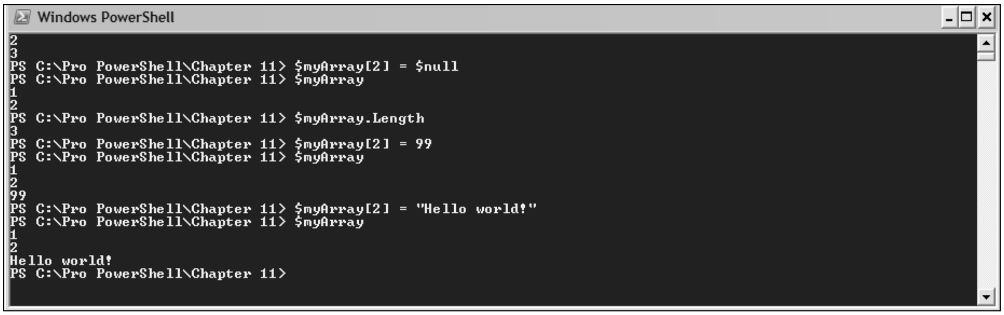


Figure 11-8

Setting the value of an element of an array to `$null` is not the same as deleting the array element, since the array element still exists. You can still specify a value for an array element that you set to `$null`. For example, you can supply a value to `$myArray[2]` by using the following command:

```
$myArray[2] = 99
```

Since the `$myArray` array is untyped you can supply a value of a different type than the value you originally removed. The following command sets the value of `$myArray[2]` to a string:

```
$myArray[2] = "Hello world!"
```

You can alter the value of any element in an array using an assignment statement. For example, you can alter the value of the second element of the `$myArray` array using this statement:

```
$myArray[1] = "I contain a string now."
```

Notice in Figure 11-9 that the value of the second element in the array has been changed. It is possible, as in this example, to replace an integer value for an element with a string value, since the array is an object array. If the array was typed, you could not change an `Int32` element to such a string.

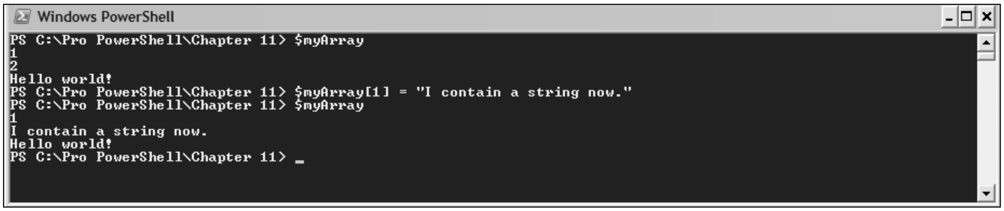


Figure 11-9

You cannot directly remove an element from an array. The length of the array is fixed when the array is created, and you cannot reduce it directly. However, you can add additional elements to an array, as shown in Figure 11-10.

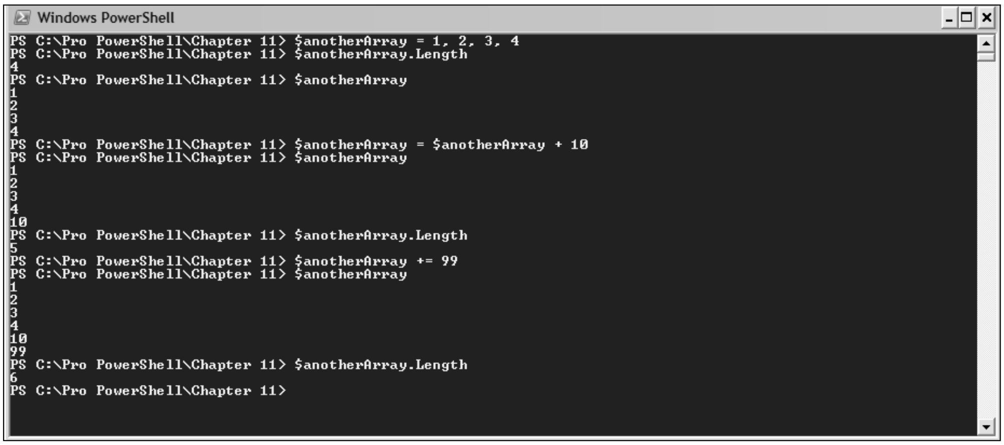


Figure 11-10

Either of the following commands adds an element to an array:

```
$anotherArray = $anotherArray + 10
```

or:

```
$anotherArray += 99
```

If you want to shorten an array, you can create a new array with a subset of the elements of an existing array. For example, to trim the last element from an existing array, `$myArray`, you can use this command:

```
$shorterArray = $myArray[0..($myArray.length-2)]
```

Remember that arrays are numbered from zero, so `$myArray.length-2` is the index of the second to last element in the `$myArray` array. The paired parentheses are essential when you use the range operator. Figure 11-11 shows this truncating a six-element array to a five-element array.

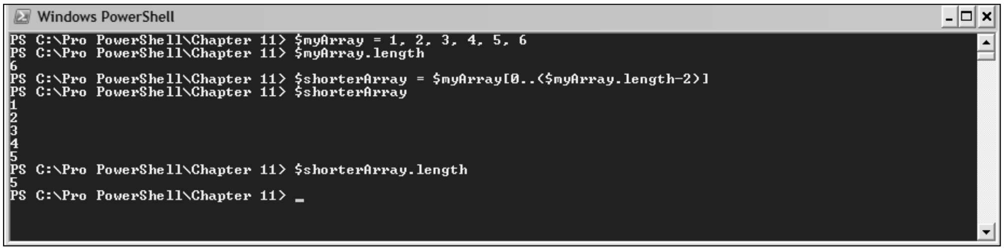


Figure 11-11

You can remove one or more elements from the middle of an existing array. For example, to remove the third element of the `$myArray` array use this command:

```
$myArray2 = $myArray[0,1 + 3..($myArray.length-1)]
```

The above command specifies that you use elements 0 and 1 followed by the range from element 3 to the end of the existing array. Notice that you must use a plus sign between the comma-separated list of element numbers and the range of elements that follows. Figure 11-12 shows the results of executing the preceding command.

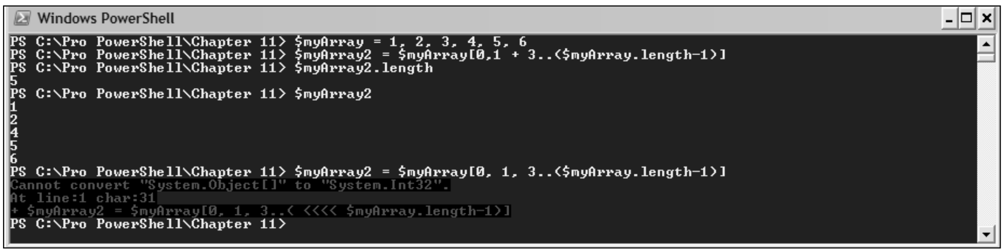


Figure 11-12

If you attempt to use syntax like this:

```
$myArray2 = $myArray[0,1,3..($myArray.length-1)]
```

you will get an error message:

```
Cannot convert "System.Object[]" to "System.Int32".
At line:1 char:31
+ $myArray2 = $myArray[0, 1, 3..( <<<< $myArray.length-1)]
```

as shown in the lower part of Figure 11-12.

You can delete an array using the `remove-item` cmdlet. For example, to delete the `$myArray` variable, type either of the following commands:

```
remove-item variable:myArray
```

or:

```
remove-item $myArray
```

Figure 11-13 shows the removal of the `$myArray` array. The former command treats the `$myArray` variable as a child item of the `variable:` drive.

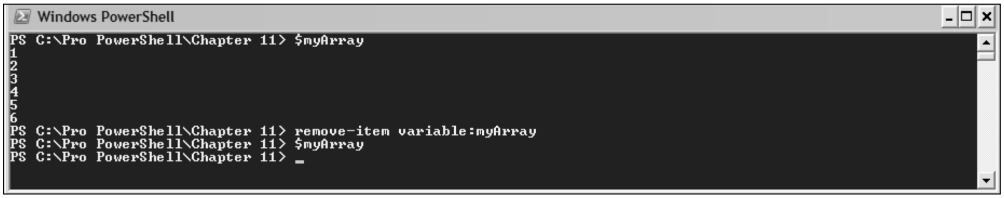


Figure 11-13

Be careful not to include the `$` sign as part of the variable name in a `remove-item` statement when explicitly using the `variable:` drive, since the `$` sign is not part of the variable's name. For example, typing

```
remove-item variable:$myArray2
```

causes Windows PowerShell to attempt to remove a variable corresponding to the values of the elements in the `$myArray2` array, and the error message is displayed as shown in Figure 11-14, which is very likely not what you intended.

```
Remove-Item : Cannot find path 'Variable:\1 2 4 5 6' because it does not exist.
At line:1 char:12
+ remove-item  <<<< variable:$myArray2
```

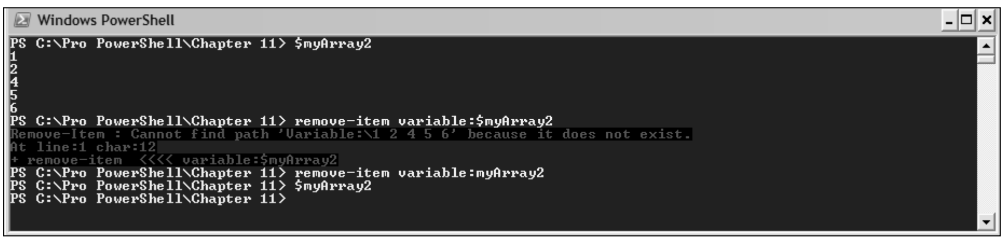


Figure 11-14

Working from the End of Arrays

Elements in Windows PowerShell arrays are numbered starting from zero. Suppose that you want to find the last four elements in an array. If you know the length of the array, say 7, you can write a command like this:

```
$myArray[($myArray.Length - 5) .. ($myArray.Length - 1)]
```

The command makes use of the `Length` property of the array. It works, as you can see in Figure 11-15, but it's a bit verbose and you need to keep in mind that array elements are numbered from zero, so you use `$myArray.Length – 1` to refer to the last element.

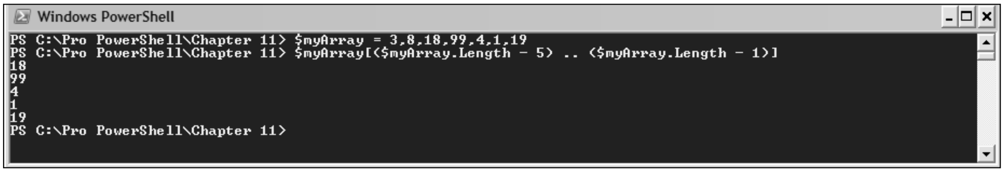


Figure 11-15

You can also change or display elements of an array starting from the last element by using negative integers as the identifier of the elements. For example, you can display the last value in an array by typing:

```
$myArray[-1]
```

To display the second last element of an array, type:

```
$myArray[-2]
```

You can display several values at the end of an array using the range operator. For example, to display the last three elements of an array with the last element displayed, first type:

```
$myArray[-1..-3]
```

If you want to display the last three elements in an array but with the third-last element displayed first, then the second-last, and so on, type this:

```
$myArray[-3..-1]
```

Figure 11-16 shows the preceding four commands used with an example array.

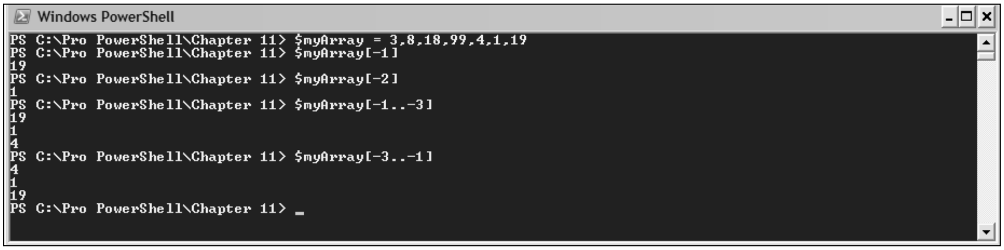


Figure 11-16

You need to be careful when attempting to mix positive and negative numbers as indices for elements. For example, if you wanted to display all but the first two elements of an array (that is, displaying the third through the last element of the array), you might type:

```
$myArray [2 .. -1]
```

But you don't get the desired result. The statement tells Windows PowerShell to display the third element, `$myArray[2]`. The range is 2 .. -1 (in other words 2, 1, 0, -1), which means that Windows PowerShell next displays `$myArray[1]`, an element that you didn't want to display, then element `$myArray[0]`, another element that you didn't want to display, and finally $myArray[-1], the last element in the array. Figure 11-17 shows the result.

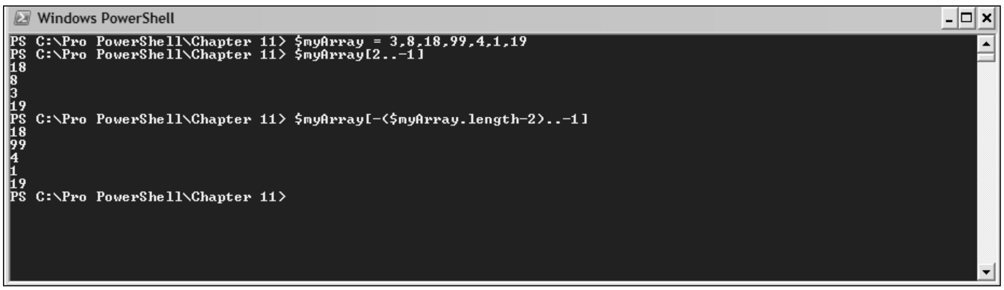


Figure 11-17

You can get the desired elements (all but the first two elements of the array) using:

```
$myArray[-($myArray.length-2)..-1]
```

Be careful with the `$myArray.length-2` part of the expression. The positively numbered array indexes start at zero but the negatively numbered array indexes start at –1. The desired result (all but the first two elements of the array) is shown in the lower part of Figure 11-17.

Using negative numbers for array elements lets you find elements counting from the end. There is another approach to do the same thing using the `select-object` element and its `–last` parameter. For example, the following command finds the last five elements of an array `$myArray`, as shown in Figure 11-18:

```
$myArray |
select-object –Last 5
```

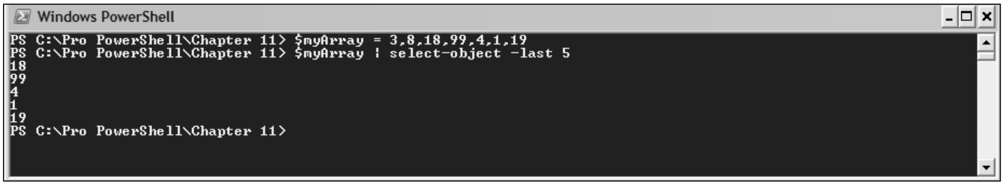


Figure 11-18

The first step in the pipeline supplies objects representing each element of the array to the second step of the pipeline. The `select-object` cmdlet finds the last five objects passed to it, as specified by the value of its `-last` parameter.

You can adapt the preceding technique to find, for example, the five largest elements in an array. Simply add a step to the pipeline using the `sort-object` cmdlet to sort the objects:

```
$myArray |
sort-object |
select-object -last 5
```

Figure 11-19 shows the five largest values in the array displayed.

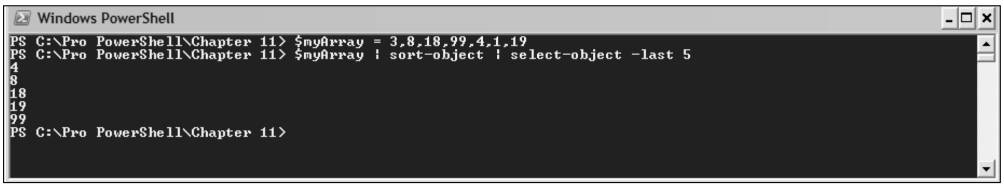


Figure 11-19

This technique, in turn, hints at how you can sort the values in an existing array. The following command sorts the values in `$myArray` and assigns those sorted values to a new array `$sortedArray`:

```
$sortedArray = ($myArray | sort-object)
```

The results are shown in Figure 11-20. It's important that you enclose the right-hand side in paired parentheses to achieve the sorted list of values.

Figure 11-20

Concatenating Arrays

PowerShell allows you to combine two arrays into a single array by concatenating the elements of the two arrays. To concatenate two arrays, you use the + operator with two existing arrays as the operands.

For example, suppose that you have already created two arrays, `$a` and `$b`. You concatenate them to create an array `$c` by typing:

```
$c = $a + $b
```

If `$a` is the array (1, 2, 3) and `$b` is the array (4, 5, 6), then `$c` is the array (1, 2, 3, 4, 5, 6).

Of course, you can concatenate them in the opposite order (with the elements of `$b` preceding the elements of `$c`), using this command

```
$c = $b + $a
```

as shown in Figure 11-21. `$c` is now the array (4, 5, 6, 1, 2, 3). The elements that come from `$b` precede the elements that come from `$a`.

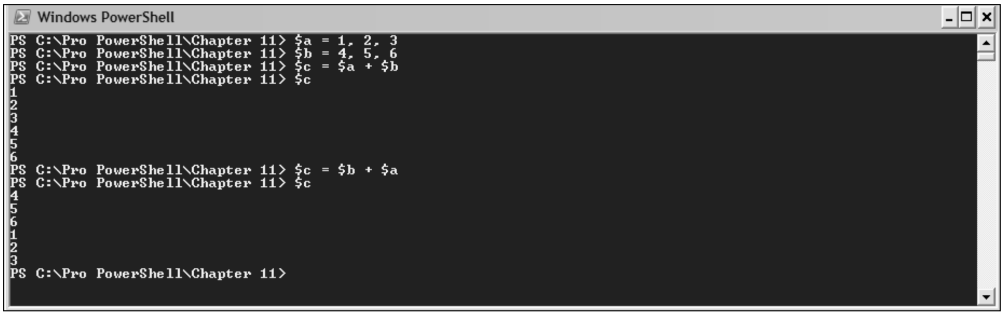


Figure 11-21

You can use the index of selected elements of each array to concatenate selected elements of two arrays. For example, to concatenate the first two elements of `$a` and the first two elements of `$b`, type

```
$d = $a[0,1] + $b[0,1]
```

as shown in Figure 11-22. `$d` is the array (1, 2, 4, 5).

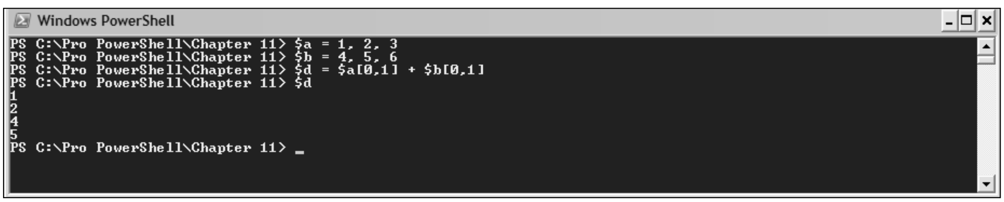


Figure 11-22

Associative Arrays

An associative array is a data structure intended to store paired keys and values. An associative array can be visualized as a two-column table, with one column holding the key and the other column holding the corresponding value in the same row. An associative array is used to store two related pieces of information. In Windows PowerShell an associative array is stored as a hash table (a `System.Collections.HashTable` object) in order to achieve good performance.

The expression which defines the key-value pairs in an associative array begins with an `@` sign and is contained between paired curly brackets. The assignment operator, =, associates a key with a value. Key-value pairs are separated by a semicolon. The following statement creates an associative array where the key is a name and the value is a date of birth:

```
$myAssocArray = @{"John Smith" = "1975/12/24"; "Alice Knowles" = "1981/03/31"}
```

As with standard arrays, to display the content of the associative array, you can simply type the name of the associative array:

```
$myAssocArray
```

Figure 11-23 shows the creation and display of the preceding associative array. As you can see in the figure, the each key-value pair is displayed on its own row.

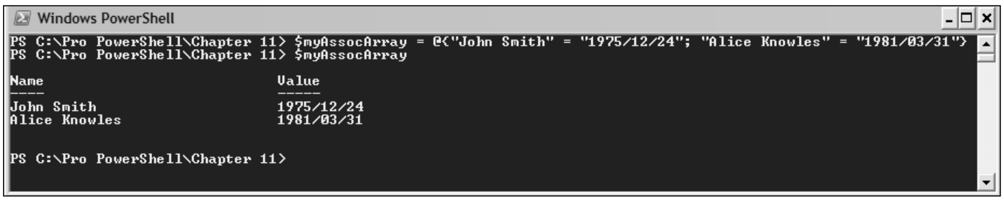


Figure 11-23

Windows PowerShell is flexible over the delimiters for both the key and value — you can use single or double quotation marks to delimit the key or value. If the key or value contains a space character, as shown in Figure 11-19, you must enclose the key or value in delimiters. Windows PowerShell uses the semicolon to separate key-value pairs. If you attempt to use a comma as separator between key-value pairs, an error message is displayed.

To selectively display the value part of a selected key-value pair, you can use an object-based notation or an array-based notation. For example, to display the value for the key John Smith, type any of the following commands:

```
$myAssocArray2."John Smith"
$myAssocArray2.'John Smith'
$myAssocArray2['John Smith']
```

Figure 11-24 shows the desired value displayed.

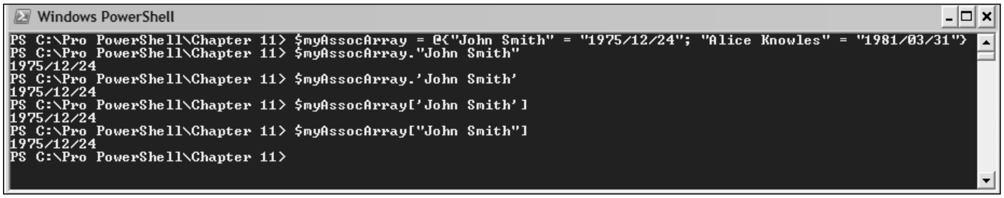


Figure 11-24

The value part of a key-value pair can be an expression. So, if you want to store a value that might be treated as an expression, for example a U.S. telephone number with area codes, you would put the value in delimiters. Otherwise, Windows PowerShell would treat the value as an expression whose value must be calculated. For example, create an associative array to contain phone numbers as follows:

```
$ExpressionArray = @{"John Smith" = 123-245-8778;
"Alice Steer"=(123)-533-2382; "Hans Allers" = "123-345-3457"}
```

Since you did not delimit the value of each key-value pair, Windows PowerShell treated the expression as a calculation, as you can demonstrate by typing:

```
$ExpressionArray
```

Figure 11-25 shows the result. To avoid this, just enclose the telephone number in delimiters, as you can see for the phone number for Hans Allers.

Figure 11-25

You can demonstrate that Windows PowerShell is treating the expression for John Smith's phone number as an `int32`, while the value for Hans Allers is set to a string, as you can see by using the following commands:

```
$ExpressionArray["John Smith"]
$ExpressionArray["Hans Allers"]
```

Conditional Expressions

Any script code beyond the most basic requires the ability to do one thing if a certain condition is specified and to do something else if the condition is not satisfied. An expression that allows such decisions to be made is called a *conditional expression*.

Windows PowerShell supports two conditional expressions:

- ❑ The `if` statement with its variants that include an `else` clause
- ❑ The `switch` statement

The if Statement

The simplest form of the `if` statement allows you to evaluate an expression, and depending on the result of the evaluation, to optionally execute a block of Windows PowerShell code.

The following script, contained in the file `ifExample.ps1`, demonstrates simple usage of a single `if` statement. The script is available for downloading from this book's web site.

```
write-host "This example shows a simple if statement in use."
$a = read-host -prompt "Enter a number between 1 and 10"
if ($a -lt 3)
{write-host '$a'" is less than 3"}
```

The `read-host` cmdlet accepts a number from the user. If the user enters a value less than `3`, the test part of the `if` statement:

```
if ($a -lt 3)
```

returns true. Therefore, the statement block (in this case a single statement) contained between the paired curly brackets, that is:

```
{write-host '$a'" is less than 3"}
```

is executed and writes the message " `$a is less than 3`" to the console. Should the test part of the `if` statement evaluate to `$false`, then the statement block is not executed. In this simple example, no code is executed if the value entered by the user is 3 or more. Figure 11-26 shows the result of running the `ifExample.ps1` script and entering a number less than 3 and a number greater than 3.

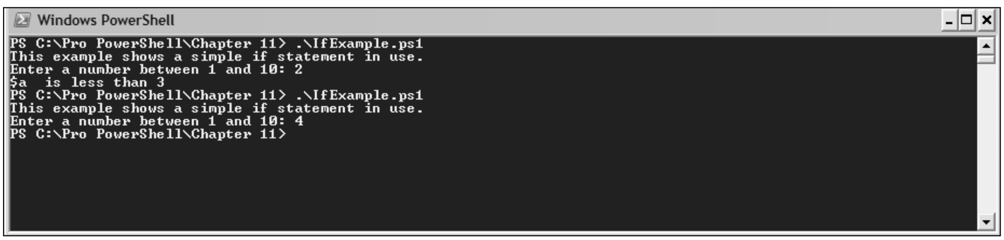


Figure 11-26

Sometimes you might want to do one thing if the test returns `$true` and something else if the test returns `$false`. The simplest form to behave in that way uses an else clause, as in the following pseudocode:

```
If (test1)
{block 1}
else
{block 2}
```

If `test1` returns `$true`, then the code in block 1 is executed. If `test1` returns `$false`, then the code in block 2 is executed.

The following code, `IfExample2.ps1`, shows how you can use the `if` statement with an `else` clause.

```
write-host "This example shows a simple if statement with an else clause."
$a = read-host -prompt "Enter a number between 1 and 10"
if ($a -lt 3)
{write-host '$a'" is less than 3"}
else
{write-host '$a'" is 3 or more"}
```

If the number that the user enters is less than three, the code in the first code block executes. If the number entered by the user is three or more, the code in the second code block executes. Figure 11-27 shows the results after entering a value of 2 (which causes the first code block to execute) and a value or 6, which causes the second code block to execute.

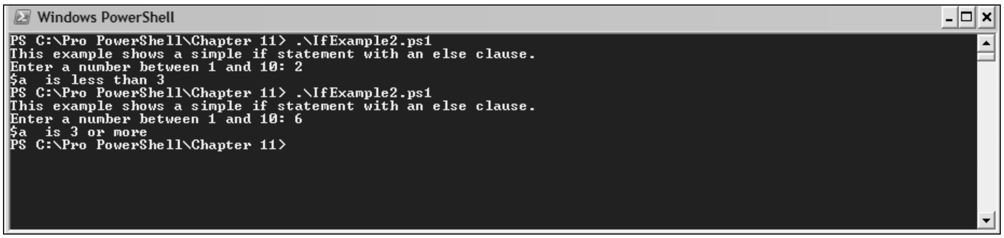


Figure 11-27

The `elseif` clause (of which there can be more than one) allows you to provide a statement block to be executed if the test in the original `if` statement returns `$false` and a later test returns `$true`. You can use multiple `elseif` clauses, as in the following pseudocode:

```
If (test1)
{block 1}
elseif (test2)
{block 2}
elseif (test3)
{block 3}
else
{block 4}
```

If `test1` returns `$true`, then code block 1 executes. Code block 2 executes only if `test1` returns `$false` and `test2` returns `$true`. Code block 3 executes only if `test1` and `test2` return `$false` and `test3` returns `$true`. If `test1`, `test2`, and `test3` all return `$false`, then code block 4 executes.

The following script, `ifExample3.ps1`, includes two `elseif` clauses that allow messages to be written to the console for any valid value (between 1 and 10) entered by a user:

```
write-host "This example shows an if statement with elseif in use."
$a = read-host -prompt "Enter a number between 1 and 10"
if ($a -lt 3)
```

```
{write-host '$a'" is less than 3"}
elseif ($a -le 5)
{write-host '$a'" is between 3 and 5 inclusive"}
elseif ($a -gt 5)
{write-host '$a'" is between 5 and 10 inclusive"}
```

If the value entered by the user is less than `3`, the first if statement returns `$true`, so the first statement block is executed:

```
if ($a -lt 3)
{write-host '$a'" is less than 3"}
```

However, if a number of `3` or more is entered, the first `elseif` clause is executed (since the first test returned `$false`):

```
elseif ($a -le 5)
```

and its test is evaluated. If that test returns `$true` (in other words $a is less than or equal to 5), the corresponding statement block:

```
{write-host '$a'" is between 3 and 5 inclusive"}
```

is executed. If, however, the test of that first `elseif` clause evaluates to false, then the second `elseif` clause is executed and its test evaluated. If that returns `$false`, then no statement block is executed. If it returns `$true`, then the statement block

```
{write-host '$a'" is between 5 and 10 inclusive"}
```

is executed. Figure 11-28 shows the result when the script is run three times with values that, respectively, return `$true` for the `if` statement, the first `elseif` clause, and the second `elseif` clause.

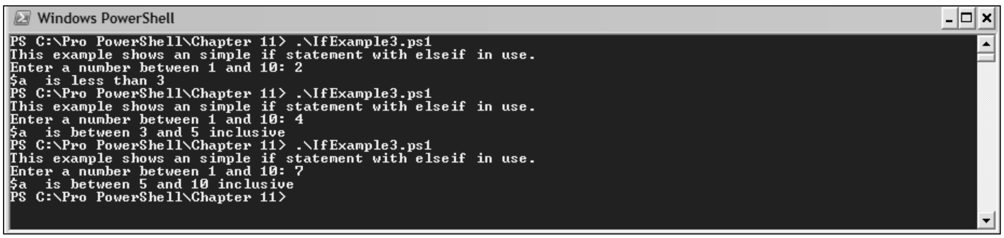


Figure 11-28

You can nest `if` statements. The following example shows two `if` statements with the second nested inside the first. The file is `IfNested.ps1`.

```
write-host "This example shows nested if statements."
$a = read-host -prompt "Enter a number between 1 and 10"
if ($a -lt 3)
{write-host '$a'" is less than 3";
 if ($a -lt 5)
 {
```

```
    write-host '$a'"is also less than 5"
   }
  }
```

Notice that the whole of the nested if statement is contained inside the opening curly brace and closing curly brace of the first `if` statement. Figure 11-29 shows the result if a number less than 3 is entered. The first test returns `$true`, so the code

```
  {write-host '$a'" is less than 3";
```

executes. The nested `if` statement is in the same code block, so it executes, too. The test `($a -lt 5)` returns true, so the code in the nested code block

```
  write-host '$a'"is also less than 5"
```

also executes.

Figure 11-29

If the first test returns `$false`, none of the code (including the nested `if` statement) in the first code block is executed.

The switch Statement

Often, you use the `if` statement described in the preceding section to test for a single result and then execute code conditionally. For example, look at the number of open handles a process has and if this is over some limit, then print out the process name). However, where you have large numbers of tests to perform on a single variable, the Windows PowerShell `switch` statement helps you to handle the situation where multiple tests are to be applied.

The simplest form of the `switch` statement is similar to the code in `SwitchExample1.ps1`, shown here:

```
  write-host "This is a very simple switch statement example."
  $a = read-host "Enter a number between 1 and 4"
  switch ($a)
  {
  1 {write-host "You entered 1"}
  2 {write-host "You entered 2"}
  3 {write-host "You entered 3"}
  4 {write-host "You entered 4"}
  }
```

The value to be tested is expressed in parentheses:

```
switch ($a)
```

and a statement block in paired curly brackets follows. Inside the paired curly brackets are a series of options against which the value of `$a` is tested. Each option has a corresponding statement block, which is executed if the value matches the value of `$a`. Figure 11-30 shows the result of executing the script multiple times, entering the values `1`, `2`, `3`, `4`, and `78`. The first four values have a match inside the statement block for the `switch` statement, so the relevant statement block containing a `write-host` cmdlet statement is executed. When the value entered is 78 there is no match for the value of `$a`, so no statement block is executed.

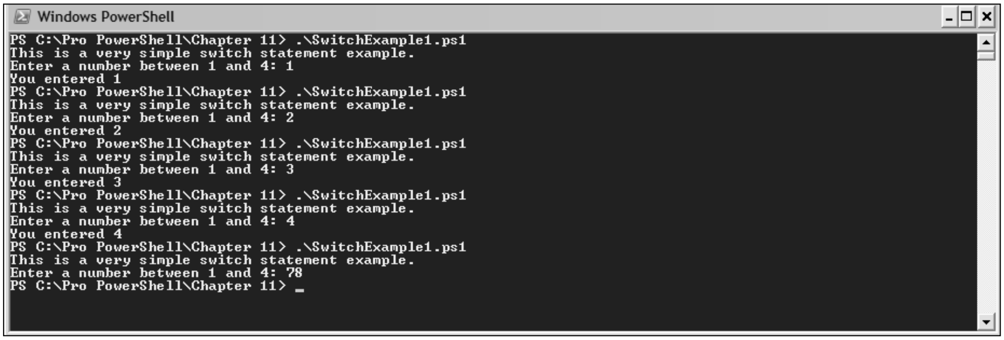


Figure 11-30

You can also provide a `default` option whose statement block is executed if no earlier option has been matched. The code in `SwitchExample2.ps1` provides a `default` option.

```
write-host "This is a very simple switch statement example."
$a = read-host "Enter a number between 1 and 4"
switch ($a)
{
1 {write-host "You entered 1"}
2 {write-host "You entered 2"}
3 {write-host "You entered 3"}
4 {write-host "You entered 4"}
default {write-host "You didn't enter a number between 1 and 4"}
}
```

Figure 11-31 shows the results when `SwitchExample2.ps1` is run.

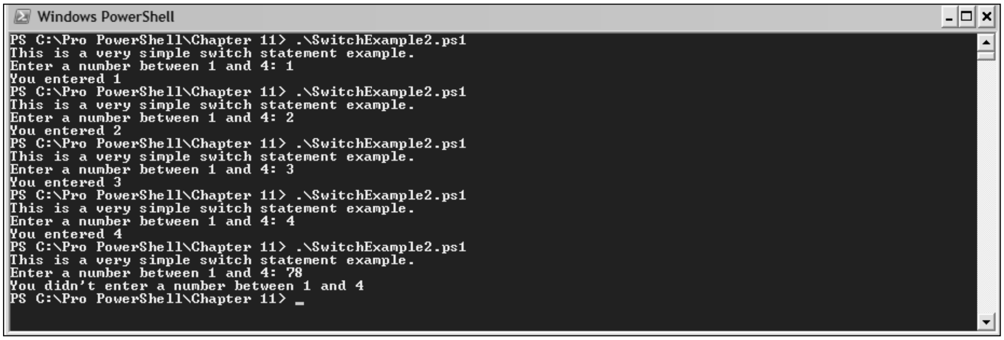


Figure 11-31

Looping Constructs

Often in scripts, you want a piece of code to be executed multiple times. You might want it to be executed a specified number of times while a particular condition is true or once for every item in a collection. Windows PowerShell supports looping constructs for each of those situations:

- `for` **loop** — Executes code a specified number of times
- `while` **loop** — Executes code while a specified condition is true
- `foreach` **loop** — Executes code for each item in a collection

Each of the preceding constructs is described in the following sections.

The for Loop

The `for` statement allows a statement block to be run multiple times, depending on a condition tested before the statement block is run. Whether or not the statement block is executed depends on the result of a conditional test. The `for` statement takes the following general form:

```
for ( initialization ; test condition; action)
{
 # a block of statements can go here
}
```

The following script, `ForExample.ps1`, shows a simple example using the for statement:

```
for ($i = 0; $i -lt 10; $i++)
{
write-host "The value of "'$i'" is $i"
}
```

The variable `$i` is initialized to `0`. The test condition is evaluated. While the value of `$i` is less than `10`, the statement block is executed. Each time the statement block is executed, the value of `$i` is incremented.

The test condition is then evaluated again. If it evaluates to `$true`, then the statement block is executed again. If it evaluates to `$false`, then the `for` loop is completed. Figure 11-32 shows the results when `ForExample.ps1` is run.

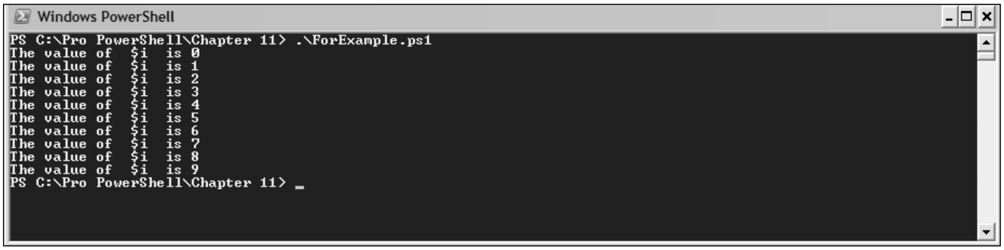


Figure 11-32

The `for` statement is very flexible, since the test condition and action can be quite complex. The condition can be any statement that evaluates to `$true` or `$false`. The action can be a simple increment, as in the previous example, or can increase according to some other basis. The script `ForExample2.ps1`, below, displays even numbers up to 20.

```
write-host "This example displays even numbers up to 20."
for ($i = 0; $i -le 20; $i+=2)
{
write-host "The value of "'$i'" is $i"
}
```

The action adds 2 to the value of `$i` each time the for statement is executed. The condition tests whether the value of `$i` is less than or equal to 20. Figure 11-33 shows the result of executing the code.

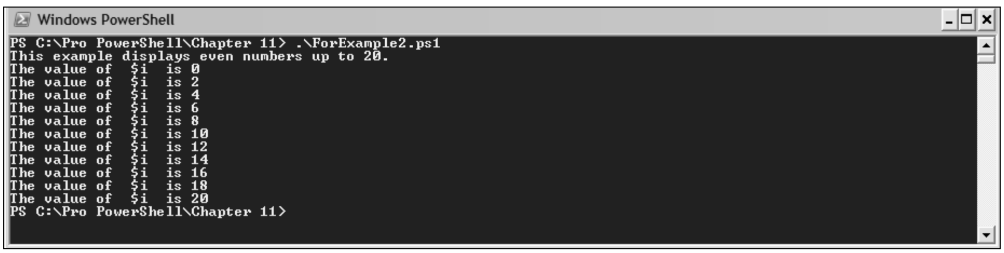


Figure 11-33

You can also use an expression to initialize the `for` loop. The following example, `ForExample3.ps1`, uses the `Month` property of a `DateTime` object to initialize `$i`. The remaining months in the year are displayed.

```
write-host "This example displays month numbers remaining in the year."
for ($i = (get-date).month; $i -le 12; $i++)
{
write-host "The value of "'$i'" is $i"
}
```

Figure 11-34 shows the result of executing the preceding code when the current month is November.

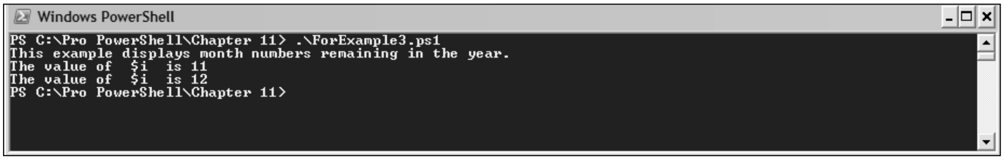


Figure 11-34

The while Loop

The `while` statement is another looping construct in Windows PowerShell. A conditional test is applied before the statement block executes. While the conditional test returns `$true`, the statement block executes. It takes the following general form:

```
while (test condition)
{
# Statement(s) to be executed
}
```

The script `WhileExample.ps1` is shown here:

```
$i = 0
while ($i  -lt 5)
{
write-host '$i'" is currently $i."
$i++
}
```

The variable `$i` is first assigned a value. Then the `while` statement executes, and the test condition specified on the following line is evaluated.

```
while ($i  -lt 5)
```

If the test condition evaluates to `$true`, the statements inside curly braces are executed.

The `while` statement itself does not provide a way to modify the value of the variable being tested inside the test condition. You need to explicitly add a statement such as

```
$i++
```

inside the statement block to avoid endless looping.

Figure 11-35 shows the result of running the `WhileExample.ps1` script.

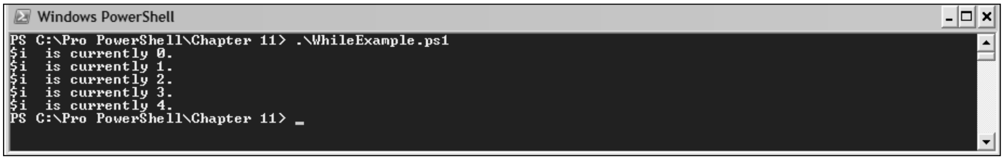


Figure 11-35

If the test condition in a `while` statement evaluates to `$false` when the `while` loop is first executed, the code in the statement block is never executed.

The do/while Loop

The `do/while` loop is similar to the `while` loop. However, the `do/while` loop is always executed at least once, since the statement block is executed before the condition in the `while` statement is evaluated. It takes the following general form:

```
do
{
# Statement(s) to be executed
}
while (test condition)
```

The script `DoWhileExample.ps1`, below, shows how the `do/while` loop can be used:

```
$i = 0
do
{
write-host '$i'" is currently $i."
$i++
}
while ($i  -lt 5)
```

Figure 11-36 shows the results of executing the script `DoWhileExample.ps1`.

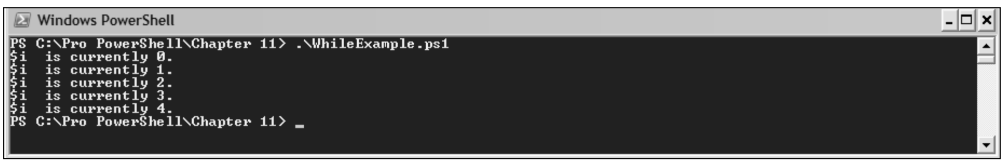


Figure 11-36

The `do/while` loop is always executed at least once, whatever the initial value of `$i`. In the following example, `DoWhileExample2.ps1, $i` would return `$false` if tested before the `do/while` loop is executed. However, the statement block runs once before the test is applied.

```
$i = 100
do
{
write-host '$i'" is currently $i."
$i++
}
while ($i  -lt 5)
```

The result of executing the preceding code is shown in the lower part of Figure 11-36.

The foreach Statement

The `foreach` statement allows you to process all items in a collection in a specified way. The `foreach` statement can, for example, be used to process all elements in an array when you don't know how many elements the array has.

The script `ForeachExample.ps1` traverses the array `$a` and displays the value of each element in a simple message written to the console:

```
$a = "a", "b", "c", "d"
foreach ($i in $a)
{
write-host "The value in the current element is $i"
}
```

Figure 11-37 shows the result of executing `ForeachExample.ps1`.

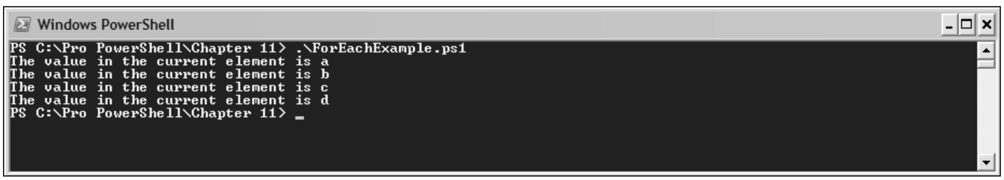


Figure 11-37

You can process a collection of objects so that one task is executed once before each object is processed, each object is then processed in the same way, and then a final statement block is executed once. The following code, `ForeachExample2.ps1`, shows an example using the `foreach-object` cmdlet to count the number of processes beginning with s.

You have to be careful how you type code like this. If you create a line that appears to PowerShell to "complete" the command, then it won't treat subsequent lines as part of the command.

```
get-process s* |
foreach-object -begin {
write-host "This is displayed in the beginning block."
$processCount = 0
} -process {

 $processCount++
```

```
} -end {
write-host "This is displayed in the end block."
write-host "The number of processes is $processCount."
}
```

The results are shown in Figure 11-38.

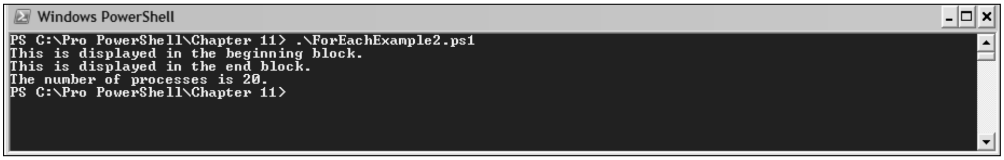


Figure 11-38

The value of the `-begin` parameter is a script block:

```
{
write-host "This is displayed in the beginning block."
$processCount = 0
}
```

Its content is executed once.

The value of the `-process` parameter is a script block:

```
{

 $processCount++
}
```

Its content is executed once for each object in the collection.

The value of the `-end` parameter is a script block:

```
{
write-host "This is displayed in the end block."
write-host "The number of processes is $processCount."
}
```

It is executed once after the collection has been processed.

You could write the code for this particular example much more simply as follows (`ForEachExample3.ps1`). The preceding example is intended primarily to show you the technique.

```
write-host "This is displayed in the beginning block."
$processCount = (get-process s*).count
write-host "This is displayed in the end block."
write-host "The number of processes is $processCount."
```

Summary

I introduced you to how arrays are expressed and manipulated in Windows PowerShell. I also introduced you to the associative array, a data structure that allows you to store collections of keys and corresponding values.

I then introduced you to the conditional expressions supported in Windows PowerShell and showed examples of how you can use them:

- The `if` statement
- The `switch` statement

Finally, I introduced you to the looping constructs supported in Windows PowerShell and showed you examples of how you can use them:

- The `while` statement
- The `for` statement
- The `foreach` statement

12

Processing Text

Windows PowerShell is designed primarily to work with .NET objects, but it also has enormous power and flexibility for the processing of text. In this chapter, I show you techniques that you can use to process text using Windows PowerShell commands and scripts.

If you have worked through earlier chapters, you will be aware that Windows PowerShell cmdlets emit objects and not strings. In that respect, Windows PowerShell cmdlets differ substantially from traditional executables such as those which form part of the traditional `cmd.exe` command shell. If you use Windows PowerShell cmdlets, you need to be able to process the objects those cmdlets emit. However, from the Windows PowerShell command shell, you can also use traditional applications that emit strings. If you use those applications and utilities and need to process the strings they emit, then the string manipulation capabilities of Windows PowerShell may be very useful. Textual data arises from many other sources, and you will likely at some time want to manipulate text using at least some of the techniques I show you in this chapter. If you commonly manipulate text data, you will use these techniques regularly when you use PowerShell.

The .NET String Class

Windows PowerShell text processing is founded on the .NET `System.string` class. As soon as you type in a string at the command line, you can access the methods and properties of the `System.String` class. For example, you can find the type of a string using this command:

```
"Hello world!".GetType()
```

You can also find the full name of the type using this command:

```
"Hello world!".GetType().FullName
```

Figure 12-1 shows the results of both commands. As you can see, the full name of the type of a string is `System.String`.

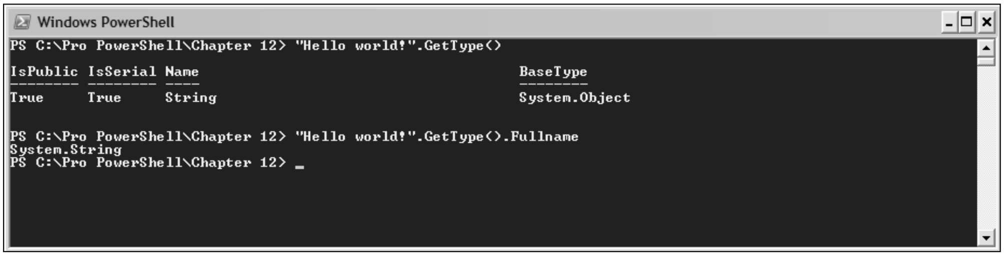


Figure 12-1

The result of the first command you typed uses Windows PowerShell's default formatter to display the result you see in Figure 12-1. If you had piped the output to the `format-list` cmdlet, you would get to see much more information about the type, as shown in Figure 12-2. That figure shows the first of several screens of information that tell you about the `String` type in the .NET 2.0 Framework. Notice that the `FullName` property is one of the many pieces of further information you can access about the object.

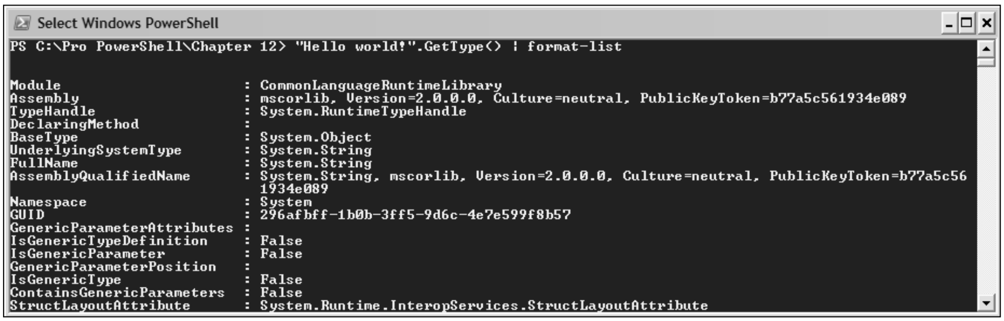


Figure 12-2

You can use Windows PowerShell to list the members of the String class. Knowing the methods of the `System.String` class allows you to explore what it is possible to do to the value of a string object.

To display the methods of the String class and page the results, use this code:

```
$Hello = "Hello world!"
$Hello |
get-member -memberType method |
More
```

The `String` class has multiple methods, which are listed in the following table. When you process strings emitted from traditional applications, you will likely need to use several of the methods of the `String` class. In the table I give the name of the method only. Typically, when using the method you write its name followed by paired parentheses (which may or may not contain one or more arguments). Thus, the `GetType` method is written with paired parentheses as follows:

```
"A string".GetType()
```

For further information on details of some methods and on underlying .NET Framework concepts consult the .NET Framework 2.0 SDK documentation.

Method	Description
`Clone`	Creates a copy of an existing string object.
`Compare`	An overloaded static method that compares two strings.
`CompareTo`	Allows you to compare one string object with another.
`Contains`	Tests whether the value of one string object contains the value of another string object.
`CopyTo`	Specifies characters to be copied from a string to a destination character array.
`EndsWith`	Tests whether the value of the string ends with a specified string.
`Equals`	Tests whether two string values are equal.
`get_Chars`	Returns a set of characters contained in a string, starting at a specified index.
`get_Length`	Returns the length of a string.
`GetEnumerator`	Enumerates individual characters in a string.
`GetHashCode`	Returns the hash code for a string.
`GetType`	Gets the type of the `String` object.
`GetTypeCode`	Gets the type code of a `String` object.
`IndexOf`	Returns the index of the first occurrence in a string of a specified character or string.
`IndexOfAny`	An overloaded method that returns the index of the first occurrence in the object instance of any character in a specified character array.
`Insert`	Inserts a string at a specified index in an existing string.
`IsNormalized`	Overloaded method that tests whether a string is normalized.
`LastIndexOf`	Finds the index in an existing string of the last occurrence of a specified character or string.
`LastIndexOfAny`	Finds the index in an existing string of the last occurrence of a character in a specified character array.
`Normalize`	An overloaded method that returns a new string object whose value is in a specified Unicode normalization form.
`PadLeft`	An overloaded method that pads the beginning of a string with a specified character to a specified length.
`PadRight`	An overloaded method that pads the end of a string with a specified character to a specified length.

Table continued on following page

Method	Description
`Remove`	An overloaded method that removes characters from a string.
`Replace`	Replaces a character with a specified character or replaces a string with a specified string.
`Split`	Splits a string into substrings. The character that defines where the string is to be split is the argument to the method.
`StartsWith`	Tests whether a string starts with the string that is the argument to the method.
`Substring`	Creates a substring from an existing string.
`ToCharArray`	Copies the characters in a string to a character array.
`ToLower`	Converts all characters in a string to lowercase.
`ToLowerInvariant`	Converts all characters in a string to lowercase, using the casing rules of the invariant culture.
`ToString`	A method inherited from `System.Object`. Since an instance of `System.String` is already a string, no conversion takes place.
`ToUpper`	Converts all characters in a string to uppercase.
`ToUpperInvariant`	Converts all characters in a string to uppercase, using the casing rules of the invariant culture.
`Trim`	Removes all occurrences of specified characters from the beginning and end of a string.
`TrimEnd`	Removes all occurrences of specified characters from the end of a string.
`TrimStart`	Removes all occurrences of specified characters from the beginning of a string.

I demonstrate the use of several of the `String` object's methods in the following sections. Those I describe and demonstrate in detail are those I anticipate will be the most useful to you in PowerShell programming.

To retrieve the properties of the any variable or array, you can use the `get-member`, as illustrated here:

```
$Hello |
get-member -memberType property
```

The `System.String` class has a single property accessible in Windows PowerShell, named `Length`. The value of the `Length` property is the number of characters in a string.

The `String` class also has a parameterized property called `Chars`, which is not exposed in Windows PowerShell.

Working with String Methods

In this section and its subsections, I describe many of the methods of the .NET 2.0 Framework `String` class. Strings in Windows PowerShell are `System.String` objects, as you saw in Figure 12-1. The methods of the `System.String` class give you great flexibility in manipulating strings in the PowerShell environment. Typically, in real life you will use several of these methods in combination to process string data in your scripts.

The Clone() Method

To create a copy of a string, you can use the `Clone()` method. The following example creates a string and assigns it to the variable `$a`:

```
$a = "Hello world!"
$c = $a.Clone()
$a
$c
$a = "Goodbye world!"
$a
$c
```

The `Clone()` method is used to copy the object and the copy is assigned to `$c`. In the third and fourth commands in the code, the value of the two strings is then shown to be the same. When you change the value of `$a`, you can see that the value of `$c` does not change. Figure 12-3 shows the results of running the preceding code.

Figure 12-3

In Windows PowerShell, instead of using the `Clone()` method you could assign one variable to the other and see behavior like that shown in Figure 12-3, which you can demonstrate by running the following code:

```
$a = "Hello world!"
$c = $a
$a
$c
$a = "Goodbye world!"
$a
$c
```

Using the Compare() Method

The `Compare()` method is an overloaded static method that allows you to compare two strings. A static method is a method available on the `System.String` class, rather than on a `System.String` object. That means that you cannot simply use the method by using the dot notation that you can use with, say, the `GetType()` method. If you attempt to execute a command such as:

```
$a.Compare($b)
```

you will see the following error message.

```
Method invocation failed because [System.String] doesn't contain a method named
'Compare'.
At line:1 char:11
+ $a.Compare( <<<< $b)
```

That doesn't mean that `System.String` doesn't have a `Compare()` method. It means that the method is only available on the `System.String` class, not on individual `System.String` instance objects.

The value returned by the `Compare()` method is an integer. A value of `0` indicates that the two strings being compared are equal. If the value is negative, then the first string is less than the second string. If the value is positive, then the first string is greater than the second string.

Since the `Compare()` method is a static method rather than a method of an individual string object, a special syntax is used:

```
[System.String]::Compare(StringObject1, StringObject2)
```

You enclose the name of the class, `System.String`, in paired square brackets. The separator between the class name and the method name is two colon characters. To compare two strings supply two literal strings or references to two string variables.

The following commands allow you to assign strings to the two variables `$a` and `$b` and compare the two string objects:

```
$a = "abc"
$b = "abcd"
[System.String]::Compare($a, $b)
$b = "ab"
[System.String]::Compare($a, $b)
$b = "abc"
[System.String]::Compare($a, $b)
```

Initially, `$a` is less than `$b` (abc is less than abcd) and that is reflected by the `Compare()` method returning `-1`. After the value of `$b` is changed, the value of `$a` is greater than the value of `$b` (abc is greater than ab); this is reflected in the return value of `1`, which is greater than `0`. Finally, the values of the two strings are set to the same sequence of characters, and the `Compare()` method returns `0`. Figure 12-4 shows the results of running the preceding commands.

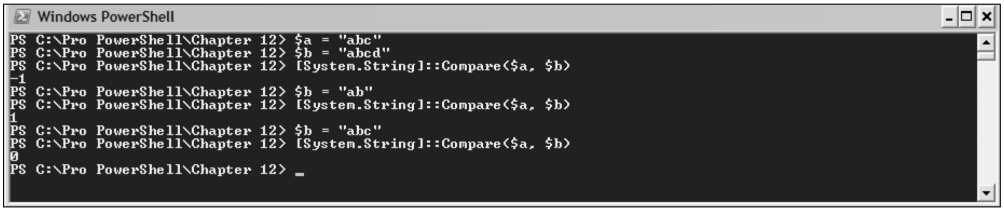


Figure 12-4

In Windows PowerShell, you don't need to use the `Compare()` method when you are comparing strings in English. You can use one of the PowerShell comparison operators. The following code performs a comparison similar to the first comparison in the previous block of code:

```
$a = "abc"
$b = "ab"
if ($a -gt $b){write-host '$a is greater than $b'}
```

You can use the `Compare()` method to compare two strings case-sensitively or case-insensitively. By default, the overload with two arguments compares strings case-sensitively, as you can demonstrate by running the following code:

```
$a = "abc"
$b = "ABC"
[System.String]::Compare($a, $b)
```

The command to carry out a specified case comparison takes the form:

```
[System.String]::(StringObject1, StringObject2, Boolean)
```

The boolean value specifies whether or not case is to be ignored when performing the comparison (the default is `$false`). Setting the boolean value to `$true` causes Windows PowerShell to ignore case and to make a case-insensitive comparison. The following commands create two strings that differ only in case. First, they are compared case-insensitively and are shown to be equal, since the `Compare()` method returns zero. When the boolean value is `$false`, the `Compare()` method indicates that the two strings are not equal, since it is false to say it's ignoring case. In other words, it is taking case into account and making a case-sensitive comparison.

```
$a = "abc"
$b = "ABC"
[System.String]::Compare($a, $b, $true)
[System.String]::Compare($a, $b, $false)
```

Figure 12-5 shows the results of running the preceding commands.

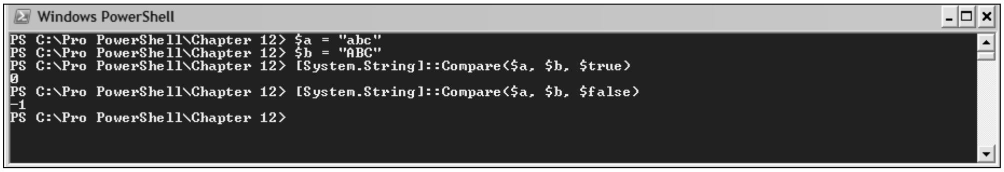


Figure 12-5

In PowerShell, you can carry out case-sensitive string comparisons using one of the case-sensitive comparison operators. For example, the code

```
$a = "abc"
$b = "ABC"
if ($a -eq $b){write-host '$a is equal to $b'}
```

displays output indicating that the strings are equal, since the comparison using the `-eq` operator in PowerShell is case-insensitive by default. However, you can make the comparison case-sensitive if the comparison is made using the case-sensitive `-ceq` comparison operator

```
$a = "abc"
$b = "ABC"
if ($a -ceq $b){write-host '$a is equal to $b'}
```

Since, when compared case-sensitively, "ABC" is not equal to "abc", the test in the `if` statement returns `$false` and the code in the `if` statement's script block is not executed. Figure 12-6 shows the result of executing the two preceding sets of commands to carry out case-insensitive and case-sensitive comparisons respectively.

Figure 12-6

There are six other forms of the `Compare()` method, which, for example, allow you to specify cultural rules and to compare parts of the string objects by using indexes into the string values. These overloads are particularly useful if you want to use cultures for languages other than English and go beyond the functionality in Windows PowerShell version 1.0. If you want to explore the definition of the `Compare()` method from Windows PowerShell use the following command, which displays the definition in a way that is a little more readable than the default.

```
([System.String] | get-member -static compare).definition.Split(',')
```

The .NET Framework 2.0 SDK has full documentation of the various overloads.

The CompareTo() Method

The `CompareTo()` method is another way to compare two strings. You can compare the value of a string object whose `CompareTo()` method is called with the value of another string object. The effect is the same as that obtained by using the `Compare()` method without a boolean argument. The `Compare()` method is a static method of the `System.String` class. The `CompareTo()` method is available on a `System.String` instance object. The `CompareTo()` method does not have the culture-specific overloads that the `Compare()` method has. By default, the `CompareTo()` method carries out a case-sensitive comparison.

The following code allows you to make comparisons, using the `CompareTo()` method, that are similar to those shown previously in Figure 12-4.

```
$a = "abc"
$b = "abcd"
$a.CompareTo($b)
$b = "ab"
$a.CompareTo($b)
$b = "abc"
$a.CompareTo($b)
```

Figure 12-7 shows the results of running the preceding code.

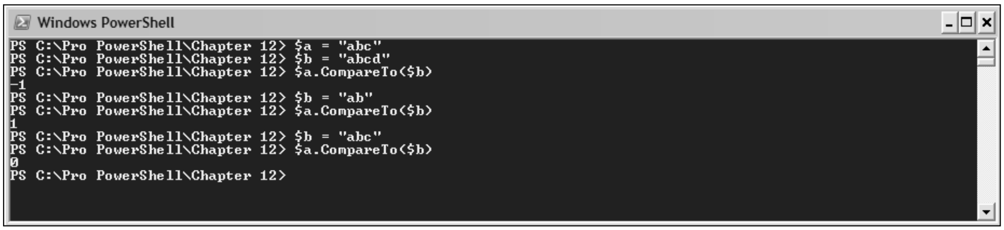


Figure 12-7

In Windows PowerShell, you can achieve results that are the same as those produced by the `CompareTo()` method, by using the PowerShell comparison operators. In fact, PowerShell gives you more flexibility, since you can carry out both case-sensitive and case-insensitive comparisons.

The Contains() Method

The `Contains()` method allows you to test whether the value of one string object contains the value of another string object. This method can be used with string literals or variables. The following code allows you to test whether the value of the variable `$a` contains a string literal and two values assigned to the variable `$b`. The `Contains()` method returns `True` when the value of the string object whose `Contains()` method is called contains the value of another string object. Otherwise, it returns `False`.

```
$a = "Hello world!"
$a.Contains("world")
$b = "ello"
$a.Contains($b)
$b = "Jello"
$a.Contains($b)
```

Figure 12-8 shows the result of executing the preceding commands.

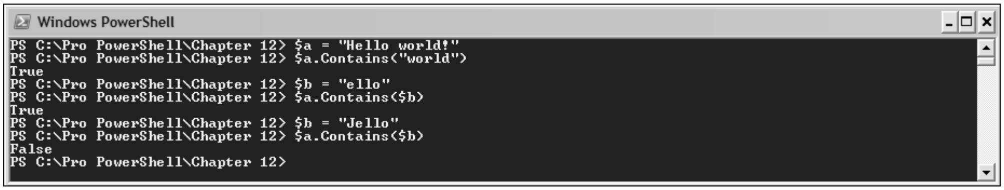


Figure 12-8

You can also use the `Contains()` method with a string literal. The following command returns True:

```
"Hello".Contains("He")
```

PowerShell provides a `-contains` comparison operator. In Release Candidate 2, it works with numeric data or character arrays. For example, the following command returns True:

```
"H", "e", "l" -contains "l"
```

But attempting to use the `-contains` operator with a string, as in the following example, returns nothing:

```
"Hello" -contains "He"
```

The CopyTo() Method

The `CopyTo()` method copies a specified sequence of characters contained in the value of a string to a specified destination in the elements of a character array. The `CopyTo()` method takes the following general form:

```
StringObject.CopyTo(IndexInString, DestinationCharacterArray, IndexInDestination,
CountOfCharactersToBeCopied)
```

The following code uses the `CopyTo()` method to copy three characters, starting at element `0` of the array `$a`, to the character array `$b`. The copied characters are copied to elements beginning at element `[2]`. The content of the character array `$b` is displayed before and after the characters are copied to it from `$a`.

```
$a = "abcdefgh"
[char[]]$b = "A", "B", "C", "D", "E", "F", "G"
$a
$b
$a.CopyTo(0, $b, 2, 3)
$b
```

Figure 12-9 shows the results when the preceding commands are executed.

Figure 12-9

The EndsWith() Method

The `EndsWith()` method tests whether the value of the string object whose `EndsWith()` method is being used ends with the string specified in parentheses. It returns a boolean value accordingly. This allows you to test if a target string returned by an application ends with a specified sequence of characters and use that test to determine how to process the string.

The `EndsWith()` method is overloaded. The simplest usage is this form:

```
StringObject.EndsWith(StringObject2)
```

The following example shows that form of the method. A string is assigned to `$a` and that string is tested separately against two test strings. The method returns, respectively, `True` and `False` for the two test strings.

```
$a = "This is a short string"
$a.EndsWith("ring")
$a.EndsWith("sentence")
```

Figure 12-10 shows the results of running this code.

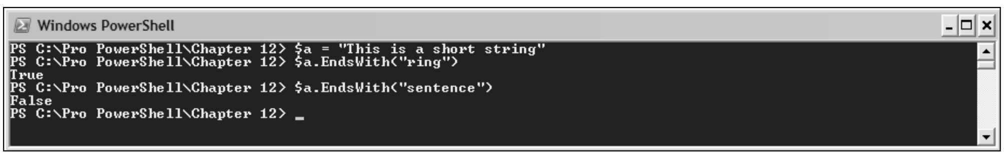


Figure 12-10

Another form of the `EndsWith()` method is the following:

```
StringObject.EndsWith(StringObject2, ComparisonType)
```

The comparison types that are new in version 2.0 of the .NET Framework are enumerated in the following list.

- `CurrentCulture` — Compares strings using culture-sensitive sorting rules defined by the current culture
- `CurrentCultureIgnoreCase` — The same as `CurrentCulture`, but case is ignored
- `InvariantCulture` — Compares strings, using sorting rules appropriate to the invariant culture
- `InvariantCultureIgnoreCase` — The same as `InvariantCulture`, but case is ignored
- `Ordinal` — Compares strings using ordinal sort rules
- `OrdinalIgnoreCase` — The same as `Ordinal`, but case is ignored

The third form of the `EndsWith()` method takes this form.

```
StringObject.Endswith(StringObject2, IgnoreCaseOrNot, CultureInfo)
```

Cultures are defined in RFC 1766 (`www.ietf.org/rfc/rfc1766.txt`). The language code is defined using the two-lowercase-character notation defined in ISO 639-1. The country/region code is defined using the two-uppercase-character notation defined in ISO 3166. Detailed discussion of culture and its handling is beyond the scope of this chapter. Further information on how the .NET Framework handles cultures can be found in the .NET Framework documentation for the `System.Globalization.Culture.Info` class.

The Equals() Method

The `Equals()` method allows you to compare two strings for equality. The `Equals()` method returns a boolean value. The method is overloaded.

The following example shows some of the ways in which the `Equals()` method can be used.

```
$a = "Hello world!"
$b = "Hello world!"
$a.Equals($b)
$c = "Goodbye world!"
$a.Equals($c)
[System.String]::Equals($a,$b)
[System.String]::Equals($a,$c)
```

Figure 12-11 shows the results of running the preceding commands.

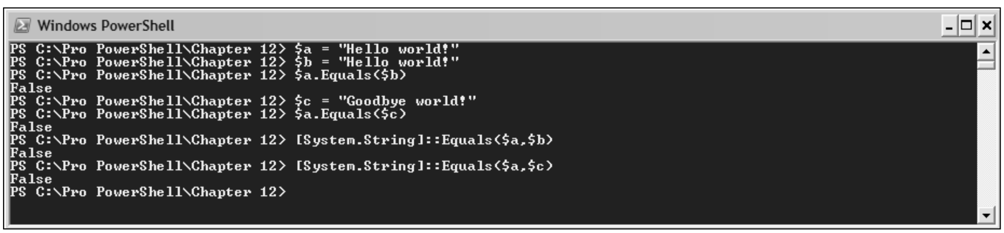


Figure 12-11

Windows PowerShell provides support for string comparisons using the `-eq`, `-ceq`, and `-ieq` comparison operators.

The get_Chars() Method

The `get_Chars()` method retrieves a character at a specified index. It takes the following general form:

```
StringObject(integerIndex)
```

The following code shows retrieval of individual characters from a sample string. The integer argument to the `get_Chars()` method corresponds to the zero-based index of the character element of the string.

```
$a = "This is a string"
$a.Get_Chars(0)
$a.Get_Chars(1)
$a.Get_Chars(2)
$a.Get_Chars(3)
```

As you can see in Figure 12-12, the indexes `0` through `3` retrieve the first four characters of the value of the string object.

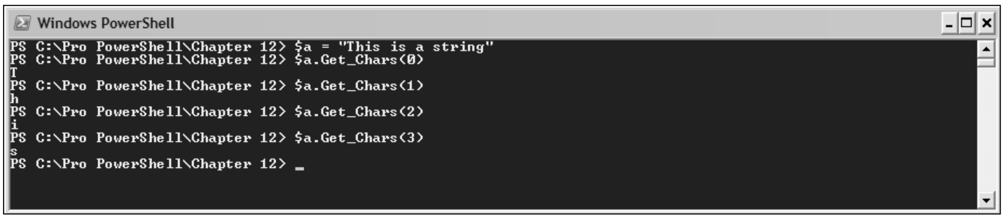


Figure 12-12

The get_Length() Method

The `get_Length()` method returns an integer value that corresponds to the number of characters in the value of the string object. It returns the same integer value as the `Length` property of the `String` object, as shown in the following example.

```
$a = "This"
$a.get_Length()
$a.length
```

Figure 12-13 shows the results of running the preceding commands at the command line.

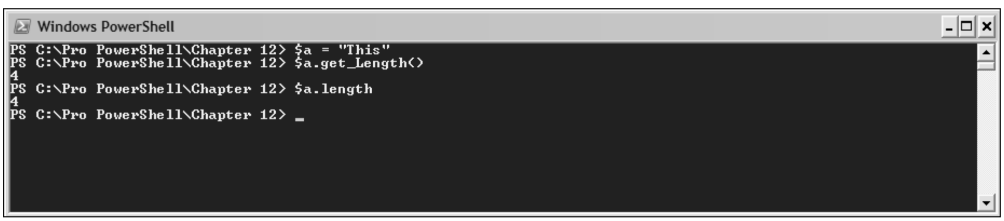


Figure 12-13

The GetType() and GetTypeCode() Methods

The `GetType()` and `GetTypeCode()` methods retrieve information about the type of an object.

To find out the type of a string, use the following code:

```
"Hello".GetType()
```

If you want to show the type of a string, together with the relevant namespace information, use the following code:

```
"Hello".GetType().FullName
```

If you only want to retrieve the type code of a `String` object use this code:

```
"Hello".GetTypeCode()
```

Figure 12-14 shows the results from running the preceding commands.

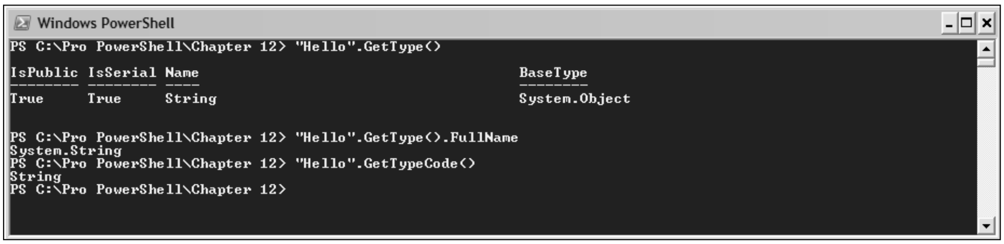


Figure 12-14

As you can see in Figure 12-14, the `GetType()` method displays, by default, a limited amount of information about the string object. If you want to see full metadata about the `String` object, use the `format-list` cmdlet as demonstrated here:

```
"Hello".GetType() |
format-list *
```

Figure 12-15 shows some of the information about the object, which is then displayed.

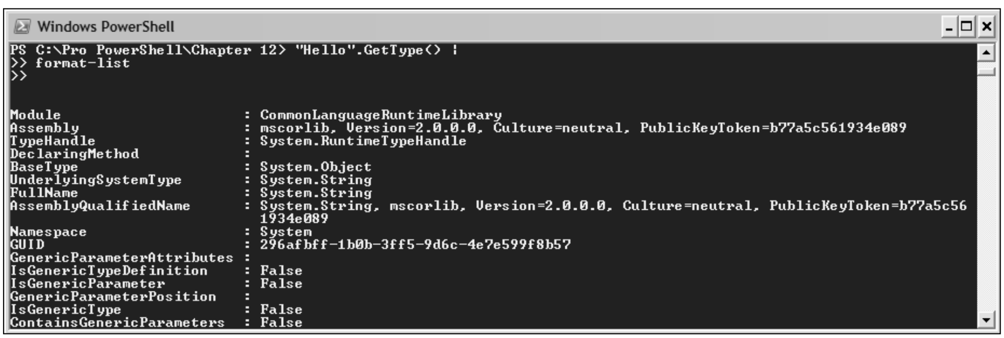


Figure 12-15

The IndexOf() and IndexOfAny() Methods

The `IndexOf()` method allows you to retrieve the index of the first occurrence in a string of a specified character or string. Once you find the index of the occurrence of a string you can use that index to capture a desired substring starting at the index. The `IndexOf()` method is an overloaded method that returns an integer value.

The following example illustrates some of the forms of the `IndexOf()` method. The first two, respectively, look for the occurrence of a single character and a string. Both return the index of `2`, since that is the index of the character sought or the first character of the specified string.

```
$a = "This is a longer sentence."
$a.IndexOf("i")
$a.IndexOf("is")
```

You can also supply an integer value specifying where the search is to start. This is shown in the following commands:

```
$a.IndexOf("i",0)
$a.IndexOf("i",3)
$a.IndexOf("i",6)
```

The first command is the same as supplying no index specifying where to start searching. The second example starts at position `3`. It therefore starts after the `i` in `This` and finds the `i` in `is`. The third command starts searching at position `6`. Since no matching character is found in the rest of the string, the value `-1` is returned.

Figure 12-16 shows the result of running the preceding commands.

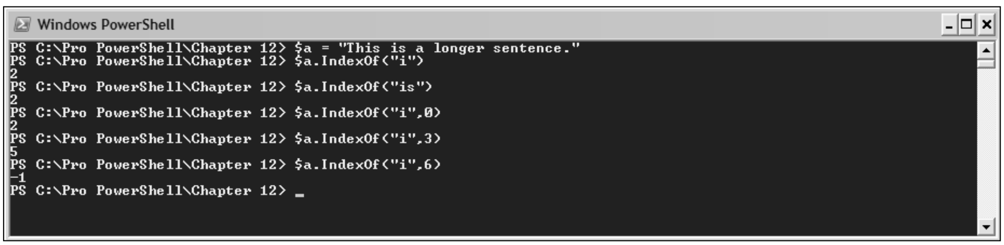


Figure 12-16

The `IndexOfAny()` method differs in that it attempts to match any character in a `char` array to the characters in a string. It reports the index of any character in the string that occurs in an element of the character array. The following example shows the use of the `IndexOfAny()` method:

```
$a = "This is a longer sentence."
[System.Char[]]$b = '.', 's', 'g'
$a.IndexOfAny($b)
```

The first character in the `char` array that occurs in the string is the character `s`, which occurs at index 3. So `3` is returned, as shown in Figure 12-17.

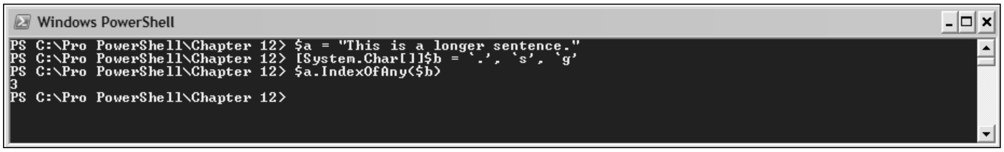


Figure 12-17

The `IndexOfAny()` method is overloaded. The other form is:

```
StringObject(char[], numberToStartAt, numberOfCharactersToSearch)
```

It allows you to search a specified substring of the string for matching characters.

The Insert() Method

The `Insert()` method inserts a specified string into an existing string, starting at a specified index. The following example shows how to use the `Insert()` method.

The aim is to insert the word PowerShell with a following space into a string that currently has the value of `Hello World!`.

```
$a = "Hello world!"
$a.Insert(6, "PowerShell ")
```

Figure 12-18 shows the result of running the code.

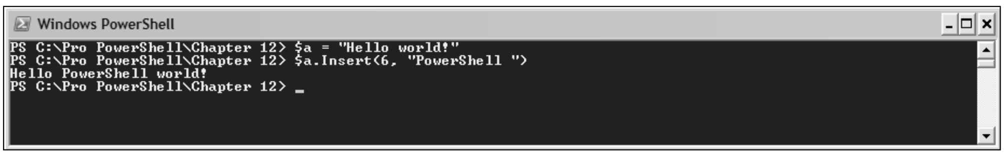


Figure 12-18

The LastIndexOf() and LastIndexOfAny() Methods

The `LastIndexOf()` and `LastIndexOfAny()` methods are used to find the last occurrence of a specified target in an existing string. The `LastIndexOf()` method finds an occurrence of a specified character or string. The `LastIndexOfAny()` method finds the last occurrence of any character in a character array. Each of the methods returns an integer value representing the index of the first character of the last match in the existing string.

The following example finds the last occurrence of the letter `l` in the string `Hello world!`. It's the `l` of `world`.

```
$a = "Hello world!"
$a.LastIndexOf("l")
```

The integer returned is `9`, which is the index of the last occurrence of the letter `l` in the string `$a`.

The following example finds the last occurrence in `$a` of any character in the character array `$b`:

```
[System.Char[]]$b = '!', 'l', 'e'
$a.LastIndexOfAny($b)
```

The integer value returned is `11`, which is the index of the exclamation mark in the value of `$a`. Figure 12-19 shows the result of running the preceding commands.

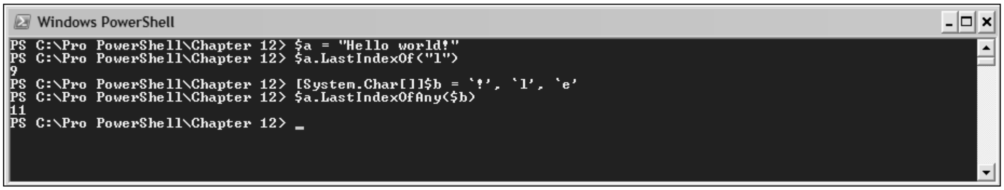


Figure 12-19

The PadLeft() and PadRight() Methods

The `PadLeft()` and `PadRight()` methods add specified characters to, respectively, the left and right of a string to increase the number of characters to a specified value. You might use these methods to adjust the position of characters in a string to correspond to a string length expected in another application or if data is to be stored in a fixed-length datatype. The methods are overloaded. When the form of the method is

```
StringObject.PadLeft(Length)
```

then the padding character is a space character, and `Length` is an integer value that is the length of the padded string. When the form of the method is

```
StringObject.PadLeft(Length, PaddingCharacter)
```

the padding character is specified explicitly.

The following example pads the string `Hello` to a length of 10 characters, using asterisks. More often you might use space characters, but they display poorly in a screenshot.

```
$a = "Hello"
$a.PadLeft(10)
$a.PadLeft(10, "*")
$a.PadRight(10)
$a.PadRight(10,"*")
```

Figure 12-20 shows the results of running the preceding commands.

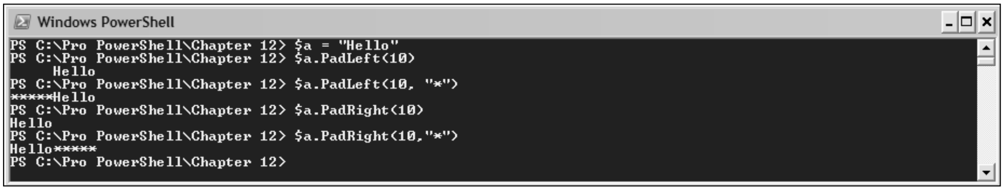


Figure 12-20

The Remove() method

The `Remove()` method removes characters from a specified string. You might use this to remove unwanted characters from the end of a string, or remove a specified number of unwanted characters from a string. The method is overloaded. The simplest form is

```
StringObject.Remove(StartingIndex)
```

which removes all characters from the specified starting index to the end of a string. The other form is:

```
StringObject.Remove(StartingIndex, NumberOfCharactersToRemove)
```

The following examples illustrate both forms of the `Remove()` method. The first use of `Remove()` removes all characters from a string, starting at index 5. Notice that the first use of the `Remove()` method produces a string that is the same length as the desired string, but the `$a` variable's length (and hence its value) is not changed. To change the value of `$a`, you need to use the second approach.

```
$a = "Hello world!"
$a.Remove(5)
$a.length
$a = $a.Remove(5)
$a
$a.length
```

Figure 12-21 illustrates the results when you execute the preceding commands.

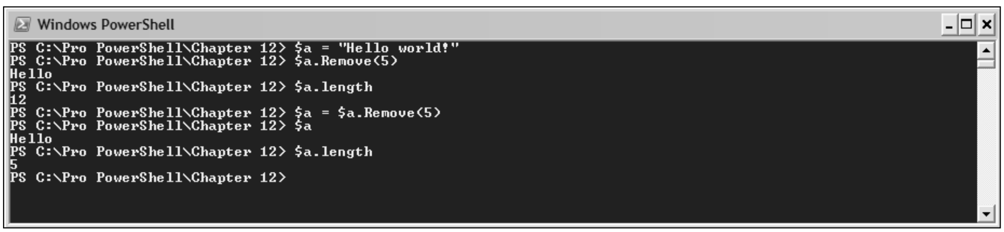


Figure 12-21

The second example shows the removal of a specified number of characters from a string. The aim is to alter the string `This is a sentence` to `This sentence` by removing characters from the string, using the `Remove()` method.

```
$a = "This is a sentence"
$a.Remove(4,5)
$a.length
$a = "This is a sentence"
$a = $a.Remove(4,5)
$a.length
```

Figure 12-22 shows the results you see when you execute the preceding commands. The first character removed is the space at the fifth position (index of 4). Five characters are removed.

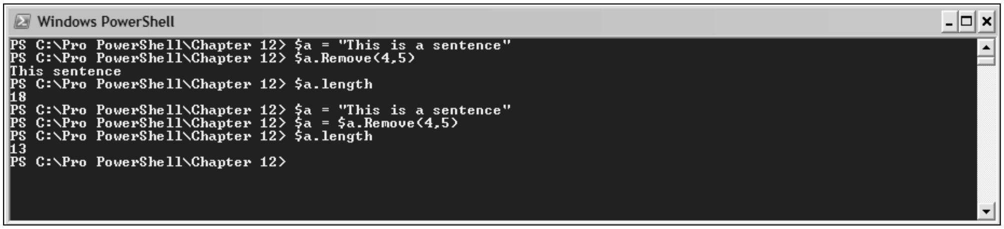


Figure 12-22

The Replace() Method

The `Replace()` method is an overloaded method that replaces all occurrences of a specified character or sequence of characters with a specified character or sequence of characters. The two forms of the `Replace()` method are shown here.

```
StringObject.Replace(oldCharacter, newCharacter)
StringObject.Replace(oldString, newString)
```

In the following example, I won't assign the result of the `Replace()` method to a variable so that I need only assign the original value to the variable once. In real code, you would likely assign the result of the `Replace()` method to an appropriate variable.

```
$a = "Mary had a little lamb."
$a.Replace('l', 'L')
$a.Replace("had", "no longer has")
```

Figure 12-23 shows the result of executing the preceding commands. The first replacement replaces twice in `little` and once in `lamb`.

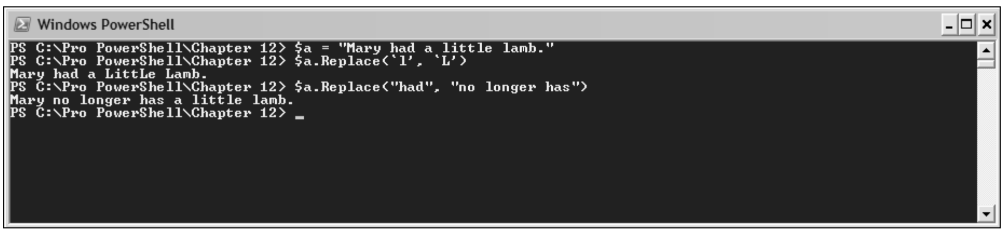


Figure 12-23

You cannot use the `Replace()` *method on a string object to use a regular expression when replacing a string. To replace a substring using regular expressions, you need to use the* `Regex` *object.*

The Split() Method

The `Split()` method, which is overloaded, allows you to split a string into substrings, using a defined delimiter character or characters (you can have multiple delimiters). The substrings are returned as a `String` array.

The following example shows how to use the `Split()` method to split a string that contains comma-separated values. The `Split()` method returns an array of the substrings contained in the original string. Notice that the delimiter character to be used as the splitting point is specified in the argument of the method. Notice, too, that the character used as the splitting point does not appear in any of the elements of the resulting `String` array. First, the string to be split is defined; then it is split using the comma as splitting point.

```
$c = "ABC,DEF,GHI,JKL"
$a = $c.Split(",")
$a
$a.GetType()
```

When the elements of the `$a` variable, which is an array, are displayed, four substrings have been assigned to the elements of the array. Figure 12-24 shows the result when you execute the preceding commands.

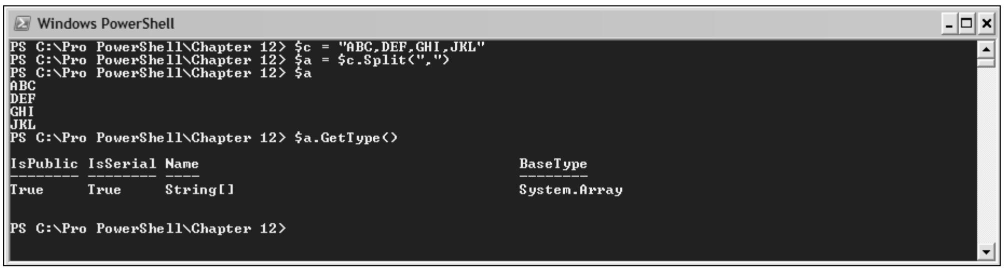


Figure 12-24

Strictly speaking, the argument to the form of the `Split()` method shown earlier is a character array — so you can specify more than one character to define the splitting point. The following example uses a comma and semicolon to split a string.

```
$c = "ABC,DEF;GHI,JKL,MNO;PQR;"
$a = $c.Split(",;")
$a
$a.GetType().Fullname
```

Figure 12-25 shows the result of executing the preceding commands.

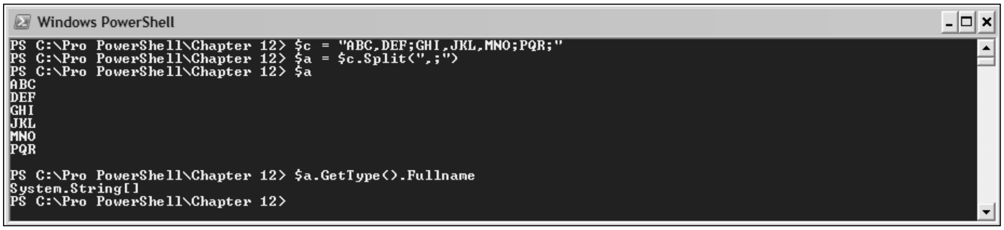


Figure 12-25

One use of the `Split()` method that is helpful when using PowerShell is to make it easier to display the definition of PowerShell's methods. Suppose that you want to find the definition of the `Split()` method itself. Use the following command to see the definition of the `Split()` method:

```
("Hello world!" | get-member split).definition
```

The part of the command inside parentheses is a two-step pipeline. A literal string is passed to the second step. In that step, the `get-member` cmdlet is used to return information about the `Split()` method. The definition property of that method is selected for display. As you can see in the upper part of Figure 12-26, the definition isn't nicely laid out for you to read. Using the `Split()` method and specifying a comma as the separator, as in the following command, lays out the definition information onscreen in a more readable way, as you can see in the lower part of Figure 12-26:

```
("Hello world!" | get-member split).definition.Split(',')
```

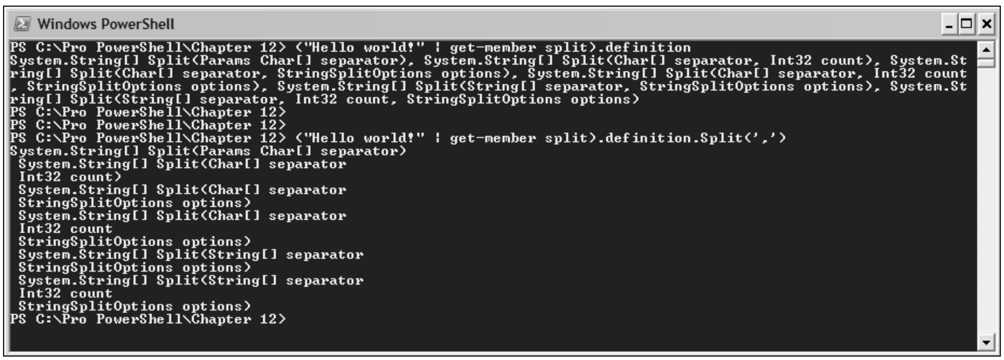


Figure 12-26

You can also use the `Split()` method to carry out a word count on a text file. You can access any of the Windows PowerShell help files, using the `$PSHome` variable. Suppose that you want to do a word count on the help file for wildcards. You can assign the content of the file to the variable `$linebyline` by using the `get-content` cmdlet to capture each line of the file, with the following command:

```
$linebyline =  get-content $PSHome\about_wildcard.help.txt
```

You can display a count of the lines using either:

```
$linebyline.count
```

or:

```
$linebyline.length
```

You can combine the lines using the following command:

```
$singleString = [System.String]::Join(' ', $linebyline)
```

The value of the variable `$singleString` is a string. You can then split the string at each space character using the `Split()` method:

```
$words = $singleString.Split()
$words.count
```

Seemingly, you have 871 words. However, if you use the following command:

```
$words |
more
```

you can see that quite a few of the "words" are blank lines. This is because there are sequences of space characters in the help file.

Figure 12-27 shows the results of executing the preceding commands.

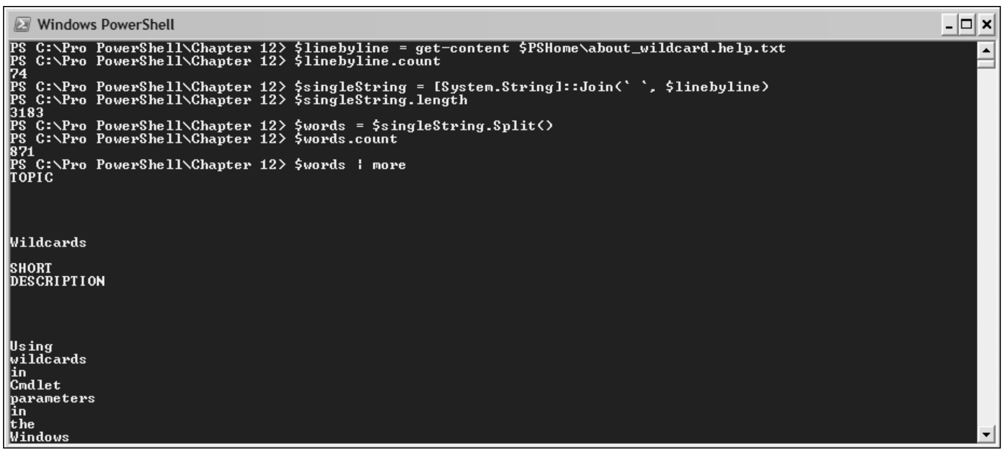


Figure 12-27

The problem of multiple successive space characters is a common one. So, the `Split()` method has a form that allows you to remove empty entries, as in the following command:

```
$wordsCleaned = $singleString.Split(' ', [stringsplitoptions]::RemoveEmptyEntries)
```

You can see that when you execute

```
$wordsCleaned.count
```

there are now only 446 words (rather than the previous 871). If you execute the following command:

```
$wordsCleaned |
more
```

and inspect the result, you can see that there are no longer blank elements of the array in `$wordsCleaned`. Figure 12-28 shows the result of executing the preceding commands.

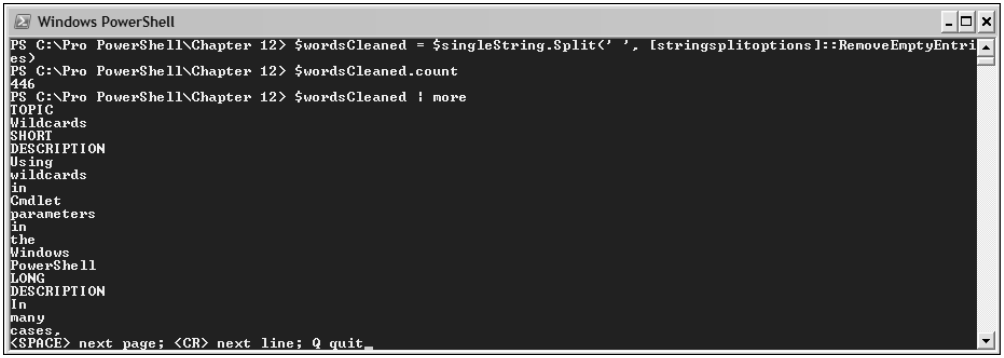


Figure 12-28

The StartsWith() Method

The `StartsWith()` method tests whether a string starts with the sequence of characters specified as the argument to the method, and returns a boolean value accordingly. The method takes three different forms:

```
StringObject.StartsWith(OtherString)
StringObject.StartsWith(OtherString,StringComparison)
StringObject.StartsWith(OtherString, IgnoreCaseOrNot, CultureInformation)
```

The enumerated values for the string comparison argument were listed earlier in this chapter in the section for the `EndsWith()` method.

The ToCharArray() Method

The `ToCharArray()` method takes the characters in a string and places one of them in each element of a character array. The method is overloaded. The simplest form of the `ToCharArray()` method is:

```
StringObject.ToCharArray()
```

The other form has two integer arguments and is:

```
StringObject.ToCharArray(StartIndex,NumberOfCharacters)
```

This takes a specified number of characters starting from the starting index in the string.

The following example shows how the `ToCharArray()` method differs from the `Split()` method. All characters are used in creating the character array when using the `ToCharArray()` method. When you use the `Split()` method, the character specified for splitting is discarded. Also, each element of the array created using `ToCharArray()` is a single character. When you use the `Split()` method, the elements of the resulting array can be strings of any length.

```
$c = "ABC,DEF;GHI"
$a = $c.ToCharArray()
$a
```

Notice in Figure 12-29 that the comma and semicolon in the existing string are included as elements in the new array when the `ToCharArray()` method is used.

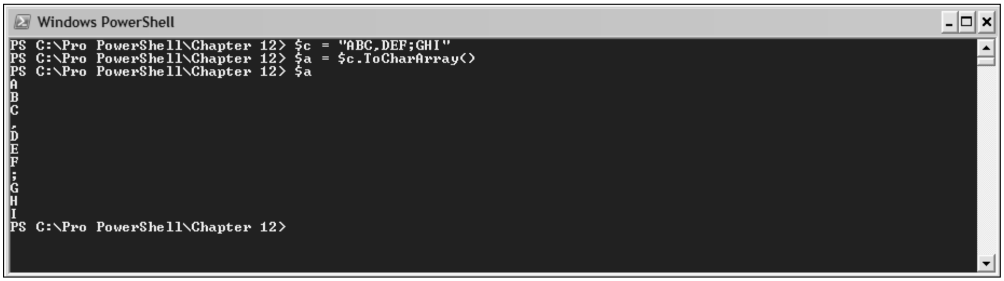


Figure 12-29

The ToLower() and ToLowerInvariant() Methods

The `ToLower()` method converts all characters in a string to lowercase. It is overloaded and takes two forms:

```
StringObject.ToLower()
StringObject.ToLower(CultureInfo)
```

The following example shows the `ToLower()` method in action:

```
$a = "Alan Smith lives in Paris."
$a.ToLower()
```

Figure 12-30 shows the results when running the preceding commands.

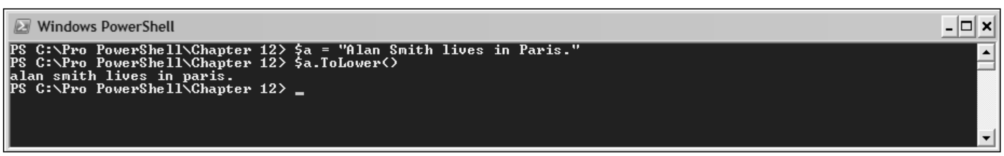


Figure 12-30

The `ToLowerInvariant()` method does the same as the `ToLower()` method but uses the casing rules of the invariant culture.

The ToUpper() and ToUpper Invariant() Methods

The `ToUpper()` method converts all characters in a string to uppercase. It is overloaded and takes two forms:

```
StringObject.ToUpper()
StringObject.ToUpper(CultureInfo)
```

The `ToUpperInvariant()` method does the same as the `ToUpper()` method but uses the casing rules of the invariant culture.

The Trim(), TrimEnd(), and TrimStart() Methods

The `Trim()` method removes specified characters from both the beginning and end of a string. The method is overloaded and has two forms. The simpler form:

```
StringObject.Trim()
```

removes whitespace characters from the beginning and end of the specified string. The other form:

```
StringObject.Trim(CharacterArray)
```

specifies an array of characters to be removed.

The `TrimEnd()` and `TrimStart()` methods are not overloaded. These methods accept a character array as their argument.

Casting Strings to Other Classes

Once you have processed strings emitted from traditional applications and suitably manipulated them you may want to explicitly convert suitable strings to other datatypes. The following sections provide several examples of doing that.

URI

Given a suitably structured string, you can create a .NET URI object by using the technique shown in this example.

```
$a = "http://www.microsoft.com"
[URI]$c = $a
```

Typing

```
$c
```

causes the properties of the URI object to be displayed, as shown in Figure 12-31.

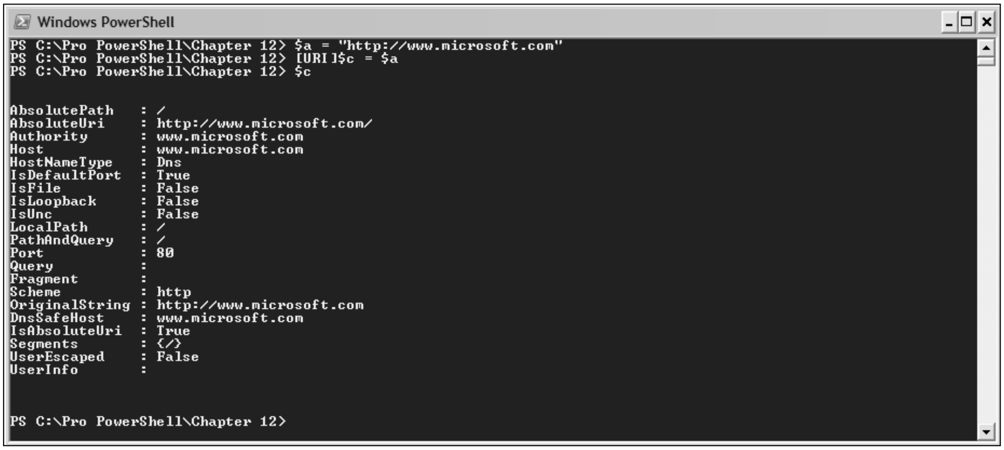


Figure 12-31

You can convert the $a variable to a URI object by using the following command:

```
$a = [URI]$a
```

datetime

If you enter a string which is a valid `datetime` value, you can easily cast it to the `datetime` datatype. That then allows you to access the properties of a `datetime` object. The following example shows how:

```
[System.Datetime]$a = "2006/12/31"
```

You can then access properties of the `datetime` object, such as the `DayOfWeek` property:

```
$a.DayOfWeek
```

Figure 12-32 shows the result of executing the preceding commands.

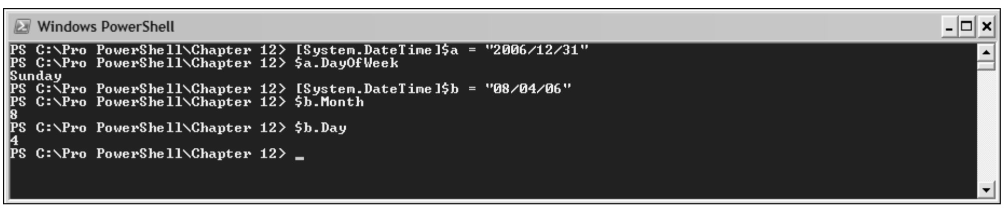


Figure 12-32

You need to be careful that you know the meaning of the string you cast to a `DateTime` object. Some dates are ambiguous, meaning one thing in one locale and having another meaning in another locale.

For example the string 08/04/2006 typically means August 4th in the United States but means 8th April in the UK. As shown in the lower part of Figure 12-32, the U.S. assumptions are used by Windows PowerShell.

To view all members of the `datetime` object, type this command:

```
$a | get-member
```

XML

A similar technique is available for casting to XML. The following example shows the creation of an XML object from a suitably structured string. All the methods and properties of the XML object are then available for use. This example uses the `get_ChildNodes()` method to display the child nodes of the document element:

```
[XML]$xml = "<Books><Book>Beginning Regular Expressions</Book><Book>Professional
Windows PowerShell</Book></Books>"
$xml.get_ChildNodes()
```

As you can see in Figure 12-33, there are two child nodes whose name is `Book`.

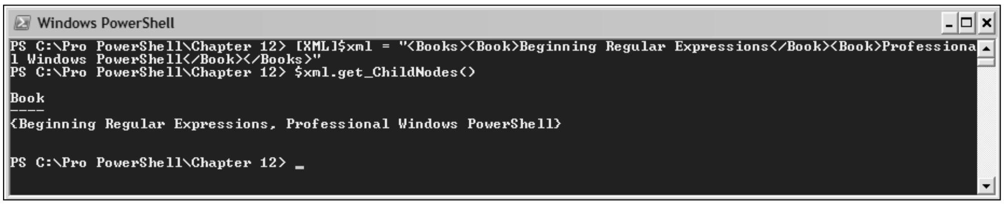


Figure 12-33

Regex

You can use the full power of .NET regular expressions to work with strings in Windows PowerShell. You use the `-match` operator when testing for a match between a string (on the left side of the `-match` operator) and a regular expression pattern (on the right side of the `-match` operator).

The following examples test for matches between a literal string and a regular expression pattern. To test if a string, `ABCDEF`, matches a pattern use either of the following commands:

```
"ABCDEF" -match ".*CD.*"
```

or:

```
"ABCDEF -match "CD"
```

Either of the preceding commands looks for the character sequence `CD` in the string. Since the string contains that pattern, the value True is displayed.

You can test for a match at the beginning of string using the `^` metacharacter. A metacharacter is a character with a special meaning in a regular expression pattern. The ^ metacharacter matches a position just before the first character of a string, so the pattern `^AB` matches a string that begins with `AB`.

```
"ABCDEF" -match "^AB"
```

You test for a match at the end of a string by using the `$` metacharacter. The `$` metacharacter matches a position immediately after the last character of a string, so the pattern `EF$` matches a string that ends with `EF`.

```
"ABCDEF" -match "EF$"
```

You can test if a character class is matched by enclosing a class of characters in square brackets. The pattern `Ban[dg]` matches a character sequence `Ban` followed by any one character in the square brackets. So, both the following statements return True:

```
"Band" -match "Ban[dg]"
"Bang" -match "Ban[dg]"
```

Figure 12-34 shows the results of executing the preceding commands.

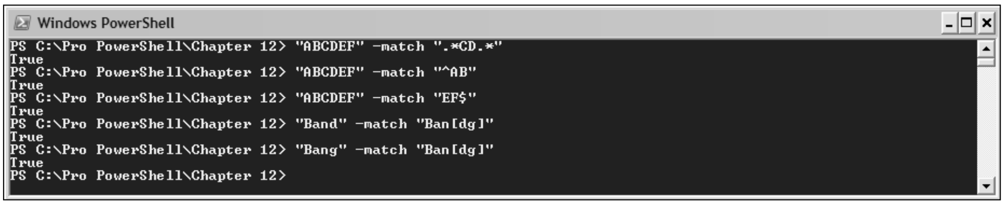


Figure 12-34

You can use the `-match` operator in the script block of the `where-object` cmdlet. Suppose that you want to find all running services that begin with the sequence of characters `SQL` in the display name. To do that, you can use the following command:

```
get-service |
where {$_.status -eq "running"} |
where {$_.displayname -match "^sql.*"} |
sort {$_.name} |
More
```

Figure 12-35 shows the results of executing the preceding text on a machine that has SQL Server 2005 installed.

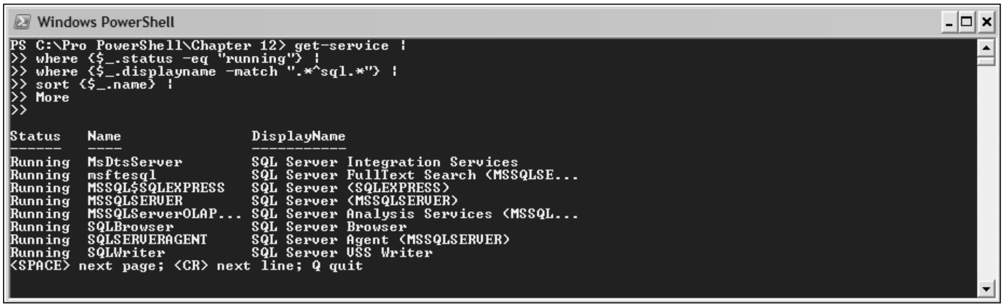


Figure 12-35

The `get-service` cmdlet without an argument will find all services. The first `where` clause:

```
where {$_.status -eq "running"}
```

filters the results so that only running services are passed on to the next `where` clause:

```
where {$_.displayname -match "^sql.*"}
```

which does the regular expression matching. The regular expression pattern, `^sql.*`, matches the position before the first character of the name, then the literal sequence of three characters, `sql`, followed by any number of characters (including zero characters). In other words, it matches any `displayname` that begins with the sequence of three characters `sql`.

Summary

Windows PowerShell is object-based but also provides powerful and flexible tools to manipulate text. The .NET `System.String` class is the basis for PowerShell's string manipulation, and this chapter discussed the methods of the `System.String` class and showed you how several of the methods are used. For some uses, Windows PowerShell provides its own syntax, for example for string comparison. For other string manipulation tasks you are dependent, at least in PowerShell version 1.0, on the methods of `System.String`. The chapter also showed you how you can convert strings to other datatypes.

13

COM Automation

Windows PowerShell is designed primarily to make use of the .NET Framework and its associated classes. However, it is going to take some time and presumably several versions of Windows PowerShell before all of a system will be exposed to Windows PowerShell as cmdlets. The Windows PowerShell team has suggested a period of 3 to 5 years for the transition to .NET and system coverage by cmdlets to be completed. Therefore, when using version 1 of Windows PowerShell and for some time afterward you are likely to want to (or have to) continue to make use of existing approaches, including the manipulation of COM objects. After all, even when there is a choice in a future version of Windows PowerShell, why should you always throw away your existing working code? The support for COM automation in Windows PowerShell means that you can leverage your existing knowledge of COM objects that are relevant to your needs.

In this chapter, therefore, I introduce you to working with COM objects from Windows PowerShell. While for some COM objects there are Windows PowerShell cmdlet equivalents, for many there are not. Well not yet anyway. If your existing COM-based code works well, you may want to postpone updating that code or writing new code.

> *Not all COM objects are supported in Windows PowerShell version 1. For example, COM objects which are based on CDO (Exchange Collaboration Data Objects) are not supported in version 1.0.*

Using the new-object Cmdlet

Windows PowerShell allows you to create and manipulate Object Linking and Embedding (OLE) automation compatible COM objects using the `new-object` cmdlet. The `new-object` cmdlet also allows you to create new .NET objects. I briefly describe the `new-object` cmdlet and then go on in this chapter to focus on its usage with COM objects. The aspects of the `new-object` cmdlet that are relevant to creating .NET objects are described in Chapter 14.

The `new-object` cmdlet can take the following parameters in addition to supporting the common parameters (which are described in Chapter 6):

- `TypeName` — The fully qualified name of the .NET type of the object that is to be created. A positional parameter used in position 1.
- `Arguments` — Arguments to the constructor of the type specified using the `TypeName` parameter. The `Arguments` parameter is optional. When present it is a positional parameter in position 2.
- `ComObject` — Used when creating a new COM object.
- `Strict` — Used with the `-ComObject` parameter. Specifies that an error should be raised if the cmdlet attempts to access an interop assembly. This allows differentiation between a true COM object and a .NET object with a wrapper.

The `TypeName` and `Arguments` parameters are used in connection with creating new .NET objects. Those parameters are described in detail in Chapter 14.

To create a new COM object, use the `new-object` cmdlet in the following form:

```
$myCOMObject = new-object -comobject NameOfCOMType
```

or, if you want to exclude the possibility of using an interop assembly, use the `-strict` parameter as in the following command:

```
$myCOMObject = new-object -comobject NameOfCOMType -strict
```

I show you working examples of this usage in the next section. However, for some COM applications, you don't need to use the `new-object` cmdlet; you can simply type the application name at the command line. For example, the following commands open an instance of Notepad and Calculator, respectively:

```
Notepad
Calc
```

However, when you start Notepad or Calculator in this way, you have no access to the object's members.

Working with Specific Applications

The following sections describe how to access a number of commonly used COM applications and carry out some tasks from Windows PowerShell. The extent of useful automation you can achieve with any individual COM object depends on the members that are exposed to you and your knowledge of those members. If you want to explore the possibilities of a particular unfamiliar COM object, use the `get-member` cmdlet to find which methods and properties you can access.

Working with Internet Explorer

Internet Explorer is one of the most easily manipulated COM applications. To launch an instance of Internet Explorer, type the following command:

```
$ie = new-object -comobject InternetExplorer.application
```

After a short pause it seems that nothing has happened, since the PowerShell cursor is displayed. No new Internet Explorer window is displayed. However, if you type a command like

```
$ie.visible
```

the value `False` is displayed on the console so you know that the `$ie` object exists and has a `visible` property. If the `$ie` object or its `visible` property didn't exist, you would expect Windows PowerShell to either display nothing (assuming that the `$erroraction` variable was set to `SilentlyContinue`) or display an error message saying that the object or property didn't exist. I discuss errors in Chapter 17.

You can explore the properties of the `$ie` object by using the command

```
$ie | get-member -memberType property
```

and the information shown in Figure 13-1 is displayed, demonstrating that the `visible` property takes a value that is of type `System.Boolean`.

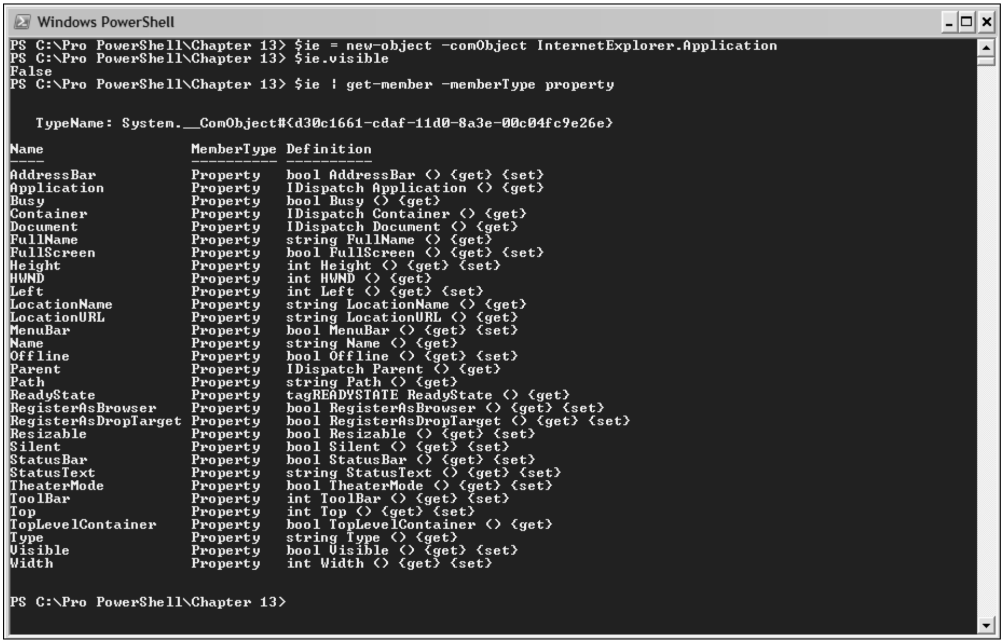


Figure 13-1

To make the Internet Explorer window visible, set the value of the `visible` property to `True`, like this:

```
$ie.visible = $true
```

Depending on how your system is set up, the Internet Explorer window will either be displayed on top of other windows or you will see an icon flashing in the start bar for the newly visible Internet Explorer window. Click the flashing icon to display the Internet Explorer window. As you can see in Figure 13-2, the Internet Explorer window doesn't display any content but does, by default, display toolbars and the status bar. Depending on what software you have installed, the appearance may vary a little. For example, I have the Google toolbar and SnagIt screen capture software installed in Internet Explorer, as you can see in Figure 13-2.

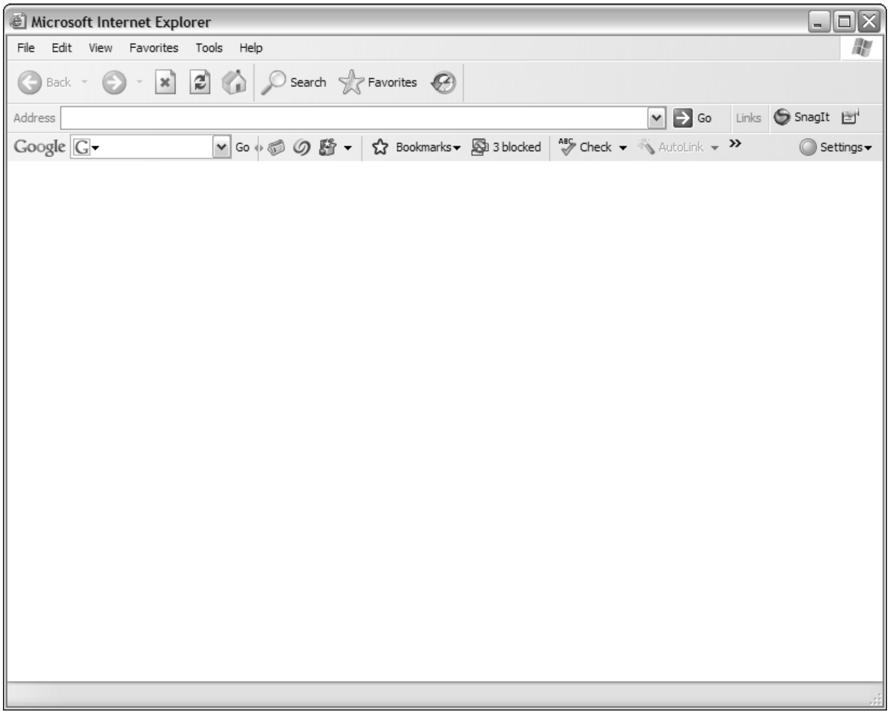


Figure 13-2

To find out what methods are available to you to manipulate or explore an instance of Internet Explorer, type:

```
$ie | get-member -memberType method
```

To see the properties grouped, use:

```
$ie | get-member -memberType Property
```

The `$ie` object has 16 methods and 31 properties, which provides significant scope for automation from Windows PowerShell.

For example, you can hide the toolbar and status bar, respectively, by using the following commands:

```
$ie.toolbar = $false
$ie.statusbar = $false
```

To make the toolbar and status bar visible again, type:

```
$ie.toolbar = $true
$ie.statusbar = $true
```

To add some text to the status bar, type:

```
$ie.statusText = "Hello Windows PowerShell World!"
```

Figure 13-3 shows the specified text displayed in the status bar.

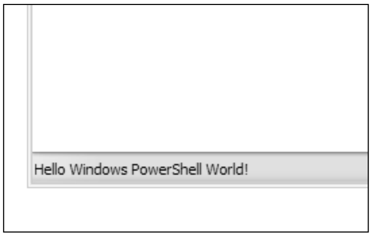


Figure 13-3

To hide and display the address bar, use the following commands:

```
$ie.addressbar = $false
$ie.addressbar = $true
```

You can set the size of the Internet Explorer window using the `height` and `width` properties. To set the window size to 200 pixels high by 300 pixels wide, use:

```
$ie.height = 200
$ie.width = 300
```

You can move the Internet Explorer window around the screen, using the `left` and `top` properties. The top-left corner of the screen is 0 from the left and 0 from the top. The following commands situate the Internet Explorer window exactly at the top left of the screen:

```
$ie.left = 0
$ie.top = 0
```

If you want to move the top left of the Internet Explorer window down, set a positive value to `$ie.top`. The larger the value, the farther down the screen the Internet Explorer window moves. If you want to move the Internet Explorer window to the right set a positive value to `$ie.left`. The larger the value, the farther the Internet Explorer window moves to the right.

You can use the `navigate()` or `navigate2()` methods to access a specified URL. For simple cases, the two methods appear to do the same thing. For example, to go to the Microsoft Web site, type:

```
$ie.navigate("http://www.microsoft.com")
```

And to go to the World Wide Web Consortium site, type:

```
$ie.navigate2("http://www.w3.org")
```

Assuming that you have entered both the preceding commands, you can go back to the Microsoft Web site using this command:

```
$ie.GoBack()
```

And then to return to the World Wide Web Consortium site, type:

```
$ie.GoForward()
```

What can you do using this facility to manipulate Internet Explorer? The following script, `DoGoogleSearch.ps1`, allows users to carry out a Google search from the Windows PowerShell command line:

```
# Carries out a Google search on a user-specified term
write-host "This Windows PowerShell script carries out a Google search on the term
you enter."
$searchTerm = read-host("Enter the desired Google search term")
$ie = new-object -comobject InternetExplorer.application
$ie.navigate2("http://www.google.com/search?hl=en&q=$searchTerm")
$ie.visible = $true
```

The `read-host` cmdlet is used to accept a search term from the user, which is assigned to the variable `$searchTerm`. Then a new Internet Explorer instance is created. Its `navigate2()` method is used to go to Google.com. The `$searchTerm` variable is used in the URL to supply one or more search terms for the search.

Figure 13-4 shows the script being executed and the user input accepted.

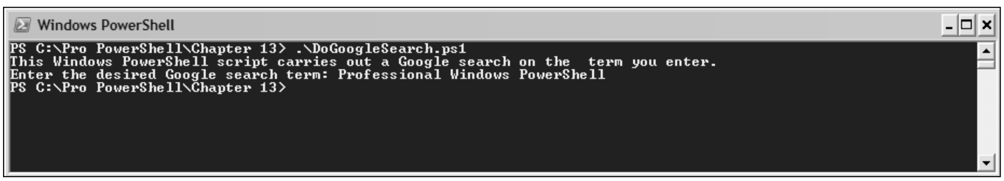


Figure 13-4

Figure 13-5 shows the Internet Explorer window that is displayed as a result of running the script shown in Figure 13-4.

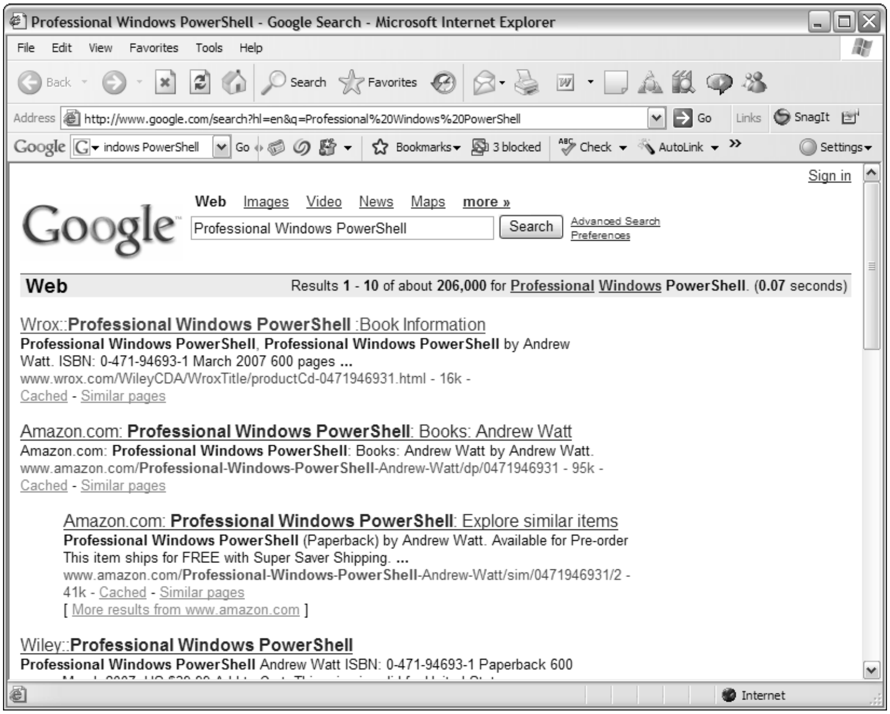


Figure 13-5

You can extend this in various ways. For example, you may want to make the search site-specific or add text to the status bar to some relevant information the user. The following code, `DoGoogleSearch2.ps1`, adapts the Google search to make the search site-specific.

```
# Carries out a site-specific Google search on a user-specified term
write-host "This Windows PowerShell script carries out a site-specific Google
search on the search term you enter."
$searchTerm = read-host("Enter the desired Google search term")
$siteToSearch = read-host("Enter the site you want to search")
$ie = new-object -comobject InternetExplorer.application
$ie.navigate2("http://www.google.com/search?hl=en&q=$searchTerm+site%3A$siteToSearc
h")
$ie.visible = $true
```

To close the Internet Explorer window from Windows PowerShell, simply type:

```
$ie.Quit()
```

Working with Windows Script Host

You can create an instance of the Windows Script Host from Windows PowerShell as follows:

```
$myWSH = new-object -ComObject wscript.shell
```

Again, nothing seems to have happened. To demonstrate that you have an instance of the Windows Script Host running, type:

```
$myWSH.popup("This pop-up window was created from Windows PowerShell!")
```

Figure 13-6 shows the pop-up window created by the preceding command.

Figure 13-6

To see the methods and properties available for Windows Script Host, use this command:

```
$myWSH |
get-member |
format-table
```

Figure 13-7 shows the members returned by the preceding command.

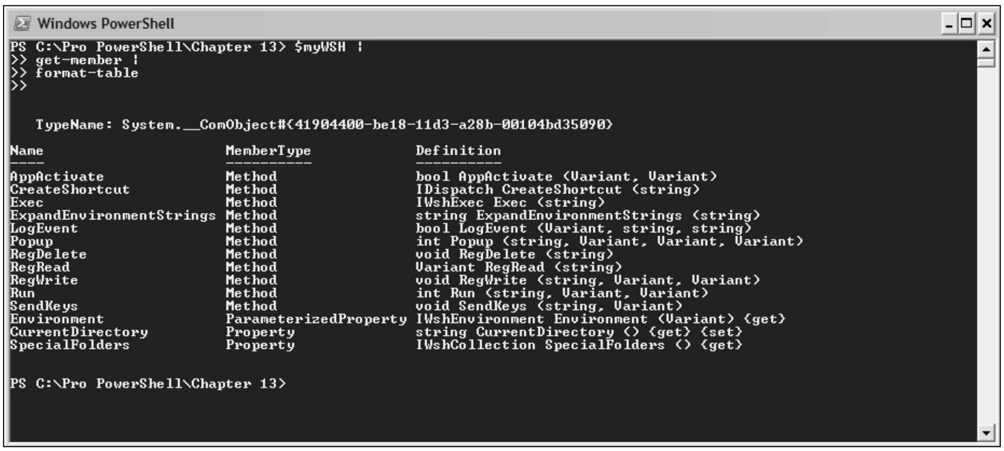


Figure 13-7

You can use the `run()` method of the Windows Script Host to run other COM applications. For example, either of the following commands will run Notepad:

```
$np1 = $myWSH.run("%windir%\System32\Notepad.exe")
$np2 = $myWSH.run("%systemroot%\System32\Notepad.exe")
```

In practice, Windows PowerShell provides a more convenient way to run executables in `C:\Windows\System32`. Simply type commands such as:

```
Notepad
```

or:

```
MSHearts
```

to launch Notepad or Hearts, respectively, from the PowerShell command line.

You can read the current `Path` environment variable by using the `RegRead()` method, as in the following command:

```
$myWSH.regread("HKLM\System\CurrentControlSet\Control\Session
Manager\Environment\path")
```

In Windows PowerShell, you have the convenience of using the command

```
get-childitem env:path
```

to access the `Path` environment variable, and you may prefer using the PowerShell Registry provider to navigate the registry.

Working with Word

To create a new instance of Word 2003 (or other installed version of Word), type this command:

```
$word = new-object -comobject Word.application
```

As with other applications, the new instance of Word is not visible when it is created. To make it visible type:

```
$word.visible = $true
```

Word has an enormous number of methods and properties available. To list them, type:

```
$word |
get-member |
out-host -paging
```

You can access various pieces of information. For example, to access your Word user name, as shown in Figure 13-8, type:

```
$word.username
```

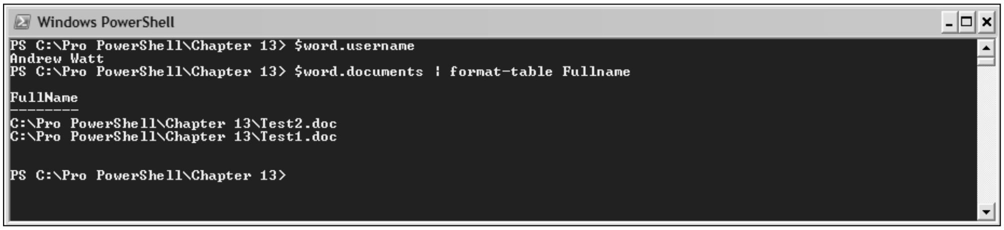


Figure 13-8

You can find the currently open documents in Words using this command, as shown in the lower part of Figure 13-8:

```
$word.documents |
format-list fullname
```

If you don't use the `format-table` cmdlet as shown, PowerShell will display an extensive list of properties of the documents. If you want to explore in detail how to work with Word documents from PowerShell, those properties will be helpful in understanding of the object model.

To find which files you have recently had open in Word, type:

```
$word.recentfiles | select-object Name
```

Working with Excel

You can work similarly with Excel. The following command creates an instance of Excel 2003:

```
$Excel = new-object -ComObject Excel.application
```

As with other COM applications, to make the relevant window visible, you need to type:

```
$Excel.visible = $true
```

To explore the many members available to you, type:

```
$Excel |
get-member
```

Assuming that you have a spreadsheet called `TestSpreadsheet.xls` in the root directory of drive c: the following script, `OpenSpreadsheet.xls`, will create an Excel object and open a named workbook. I have made the variables global so that you can access them from the Windows PowerShell command line, if you wish.

```
$global:src = "C:\TestSpreadsheet.xls"
$global:excel = New-Object -COM Excel.Application
$global:ci = [System.Globalization.CultureInfo]'en-US'
$global:book = $excel.Workbooks.PSBase.GetType().InvokeMember(
    'Open', [Reflection.BindingFlags]::InvokeMethod, $null,
$excel.Workbooks, $src, $ci)
$excel.visible = $true
```

Assuming that the script is in the current directory, type the following command to run it:

```
.\OpenSpreadsheet.ps1
```

You can find which workbook(s) are open in that instance of Excel by using this command:

```
$Excel.workbooks |
select-object Name
```

Figure 13-9 shows the result of running the preceding command when one workbook is open.

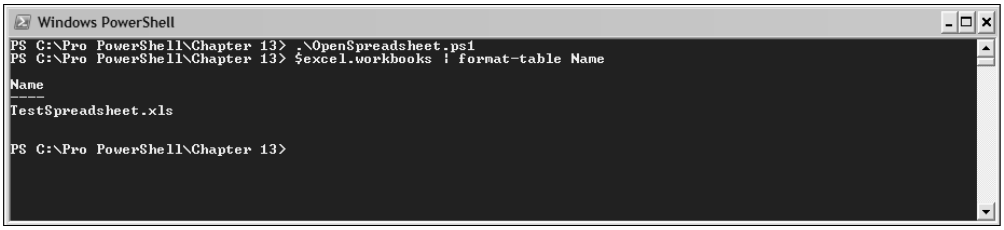


Figure 13-9

Move the cursor manually to cell B3. You can find the current active cell by using this command:

```
$Excel.ActiveCell |
format-table Column,Row -auto
```

Figure 13-10 shows the output from the preceding command when cell B3 is the active cell. Notice that the identifier for the column returned by this command is a number, not the letter you are used to in the Excel graphical user interface.

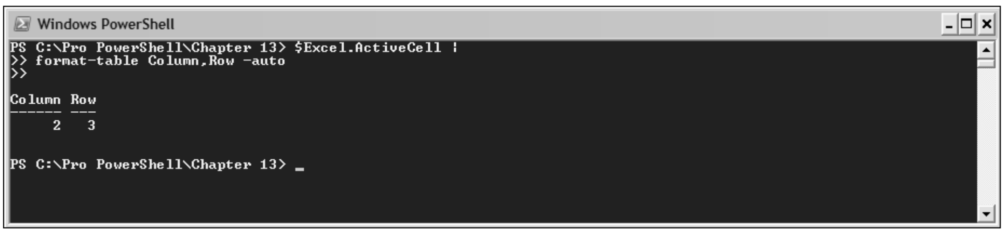


Figure 13-10

Accessing Data in an Access Database

You can access data in an Access database from the PowerShell command line. The following command shows a demonstration of what can be done. I will show you the results and explain how the code works.

```
$adOpenStatic = 3
$adLockOptimistic = 3

$global:objConnection = New-Object -comobject ADODB.Connection
```

```
$global:objCommand = "Select * from Employees"
$global:objRecordset = New-Object -comobject ADODB.Recordset

$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; Data Source =
c:\Northwind.mdb")
$objRecordset.Open($objCommand, $objConnection, $adOpenStatic, $adLockOptimistic)

$objRecordset.MoveFirst()

do {write-host $objRecordset.Fields.Item("EmployeeID").Value  " -NoNewLine;
    write-host $objRecordset.Fields.Item("LastName").Value -NoNewLine;
    write-host ","$objRecordset.Fields.Item("FirstName").Value;
    $objRecordset.MoveNext()} until
    ($objRecordset.EOF -eq $True)

$objRecordset.Close()
$objConnection.Close()
```

The preceding code assumes that you have the Northwind database in the file `C:\Northwind.mdb`. If necessary, adapt the location or name of the Northwind database. Figure 13-11 shows the results of running the script from the command line.

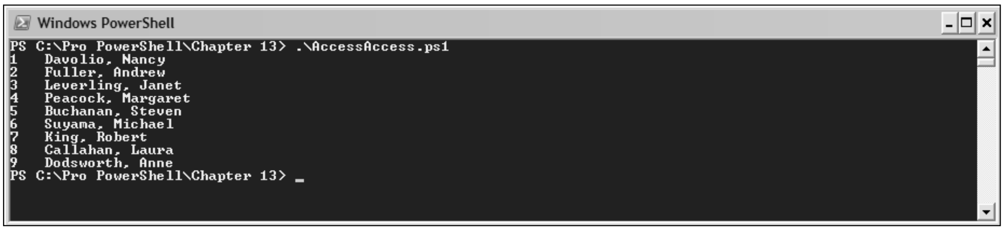


Figure 13-11

The code first sets up variables that are arguments to the method that opens a recordset later in the code:

```
$adOpenStatic = 3
$adLockOptimistic = 3
```

Next, variables are created for the connection, command and recordset objects that allow you to retrieve data from the Access database. The connection and recordset are COM objects created using the `new-object` cmdlet.

```
$global:objConnection = New-Object -comobject ADODB.Connection
$global:objCommand = "Select * from Employees"
$global:objRecordset = New-Object -comobject ADODB.Recordset
```

Next open the connection:

```
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; Data Source =
c:\Northwind.mdb")
```

Then open the recordset, using the command and connection variables as arguments:

```
$objRecordset.Open($objCommand, $objConnection, $adOpenStatic, $adLockOptimistic)
```

Move to the first record:

```
$objRecordset.MoveFirst()
```

The loop through the records:

```
do {write-host $objRecordset.Fields.Item("EmployeeID").Value"  " -NoNewLine;
    write-host $objRecordset.Fields.Item("LastName").Value -NoNewLine;
    write-host ","$objRecordset.Fields.Item("FirstName").Value;
    $objRecordset.MoveNext()} until
    ($objRecordset.EOF -eq $True)
```

The `write-host` cmdlet is used to display three values from each row in the table. Once you have displayed the desired data from the row (I chose Employee Id, last name, and first name for this example), you use the `MoveNext()` method to move on to the next row. As you can see in Figure 13-11, PowerShell isn't particularly convenient for displaying this data.

Finally, tidy up the objects you used:

```
$objRecordset.Close()
$objConnection.Close()
```

Working with a Network Share

You can map a drive on a network share to a drive on the local machine. First, create a variable that uses a COM `WScript.Network` object:

```
$network = new-object -comObject WScript.Network
```

One of the methods you can see if you use the command

```
$network | get-member
```

is the `MapNetworkDrive()` method. You can use that to map a fileshare on a remote machine to a drive on the local machine. The following command does that:

```
$network.MapNetworkDrive("L:", "\\Helios\Documents")
```

You can then switch to the newly created mapping:

```
L:
```

And view its contents:

```
get-childitem
```

Figure 13-12 shows the results of executing the preceding commands to connect to a network share on one of my development machines.

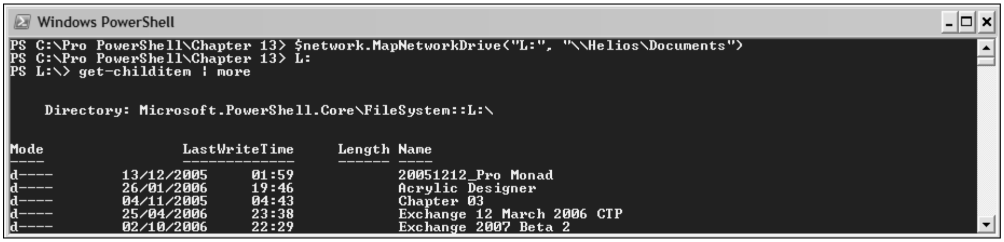


Figure 13-12

You can also see that the drive has been added using the `get-psdrive` cmdlet.

```
get-psdrive L*
```

You can use the `RemoveNetworkDrive()` method to remove the mapping to the network drive:

```
$network.RemoveNetworkDrive("L:")
```

You can see in Figure 13-13 that the drive L: is no longer available.

Figure 13-13

Using Synthetic Types

The .NET type system that Windows PowerShell uses is extensible. This extensibility is directly relevant to how Windows PowerShell handles COM objects.

All COM objects have the same type, `System.__ComObject`, in a .NET Framework setting. That causes potential problems for Windows PowerShell in distinguishing one COM object from another, since although all COM objects have the `System.__ComObject` type, there is a huge variety of underlying functionality and a large range of members of individual object instances. If you can't unambiguously identify the kind of object you are dealing with, it makes writing code to use methods or access or manipulate properties highly problematic. To resolve that ambiguity, when creating a COM object Windows, PowerShell creates a *synthetic type* that uses the common `System.__ComObject` type and adds a # to it followed by the Class ID stored in the registry for the relevant class.

The following example displays the synthetic type for Internet Explorer. Create a new instance of Internet Explorer, using the command:

```
$ie = new-object -ComObject InternetExplorer.application
```

Then use the following command to display the members of the `$ie` object:

```
$ie |
get-member
```

A lengthy list of members is displayed, but before those the synthetic type, which refers to the registry class for Internet Explorer, is displayed. Figure 13-14 shows the synthetic type for Internet Explorer. It is displayed as `System.__ComObject#{d30c1661-cdaf-11d0-8a3e-00c04fc9e26e}`. This represents `System__ComObject`, plus the information from the registry.

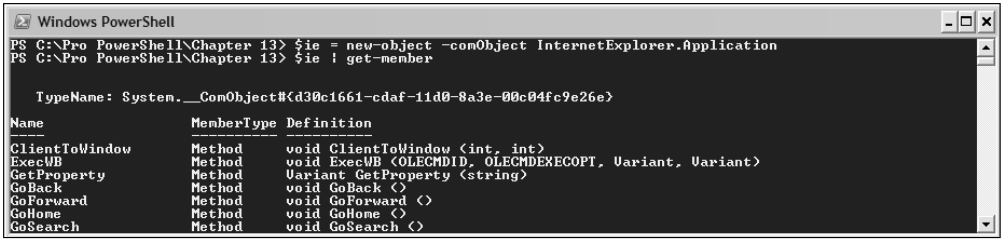


Figure 13-14

To find the relevant Registry key, use the following command once you navigate to `HKLM:\Software\Classes\Interface` in the Registry.

```
get-childitem |
where-object {$_.Name -match "d30c1661"}
```

Notice that the value used with the `where-object` parameter matches the value shown in Figure 13-14 for the synthetic type of the Internet Explorer object.

Figure 13-15 shows the relevant key for Internet Explorer at `HKLM:\Software\Classes\Interface` accessed from the Windows PowerShell command line.

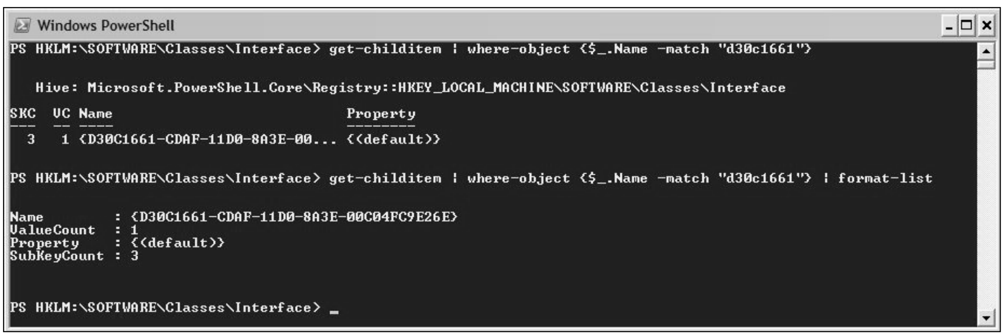


Figure 13-15

Alternatively you can access the relevant key using the `RegEdit` utility. To run `RegEdit`, click the Start, then select run and type `RegEdit` in the text box. A convenient way to access the key is to press Ctrl+F to open the Find dialog box and enter `d30c1661`. After a significant pause, `RegEdit` will stop at the desired place. Figure 13-16 shows the appearance in `RegEdit`.

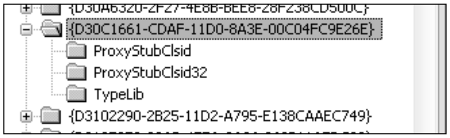


Figure 13-16

Each kind of COM object has its own unique (synthetic) type that allows you to predictably use the functionality and data available without the risk of trying to access nonexistent methods or properties, (assuming that you are familiar with the members of the object).

Summary

You can use the PowerShell `new-object` cmdlet with the `-comObject` parameter to create and automate COM objects.

This chapter showed you how you can create an Internet Explorer window and, from the PowerShell command line, navigate to a desired URL. You learned how you can create scripts using COM automation to make use of Internet Explorer. You also saw examples illustrating how you can create Google searches from the Windows PowerShell command line.

You can automate Microsoft Word and Microsoft Excel by using COM automation from PowerShell. You saw how you can access data held in an Access database, as illustrated in an example.

You also learned how Windows PowerShell creates synthetic types to distinguish the type of different kinds of COM objects.

14

Working with .NET

Windows PowerShell is based on and leverages extensively version 2.0 of the .NET Framework. This means that you can use Windows PowerShell for an enormous range of scripting functionality, taking advantage of the huge range of.NET Framework classes to provide the functionality you need to create custom scripts.

Limitations in version 1.0 of Windows PowerShell tend to reflect the fact that many facets of the Windows operating system are not yet exposed as managed objects. It is likely that an increasing proportion of a Windows system will be exposed as .NET objects in future versions of the .NET Framework and, subsequently, in future versions of Windows PowerShell. In the meantime, Windows PowerShell can be used with existing technologies where .NET classes aren't available. In Chapter 13, for example, I showed you how to work with COM objects from Windows PowerShell.

Windows PowerShell and the .NET Framework

Unless you have jumped straight into this chapter, you already know that Windows PowerShell is founded on the .NET Framework and that cmdlets and the objects passed along a Windows PowerShell pipeline are .NET objects.

Windows PowerShell provides syntax that allows you to create .NET objects and then explore the members of those .NET objects. You have seen many examples in earlier chapters of using the `get-member` cmdlet to explore the members of objects. In this chapter, I introduce you to using the `new-object` cmdlet to create new .NET objects. Using such techniques for creating and exploring .NET objects, you can create many useful scripts by combining them with other aspects of Windows PowerShell functionality.

To be able to get full advantage from the .NET functionality of Windows PowerShell, you need to have a good understanding of the parts of the .NET Framework 2.0 classes that are relevant to your needs. The scope of the .NET Framework 2.0 is huge, so I can only illustrate in this chapter the kind of things that you can do.

There are several sources of information on the .NET Framework 2.0. Two useful sources of information are Visual Studio 2005 help and the .NET Framework 2.0 Software Developer's Kit.

If you have access to an edition of Visual Studio 2005, you can access large quantities of useful information on the .NET Framework 2.0. With Visual Studio 2005 open, select Help ➪ Contents. After a pause the Microsoft Visual Studio 2005 Documentation opens in Microsoft Document Explorer. The volume of information can be overwhelming. Pin the Contents pane open, and select .NET Framework from the Filtered by dropdown in the Contents pane. Figure 14-1 shows the appearance with some nodes in the filtered data expanded. Notice that the Class Library Reference is highlighted.

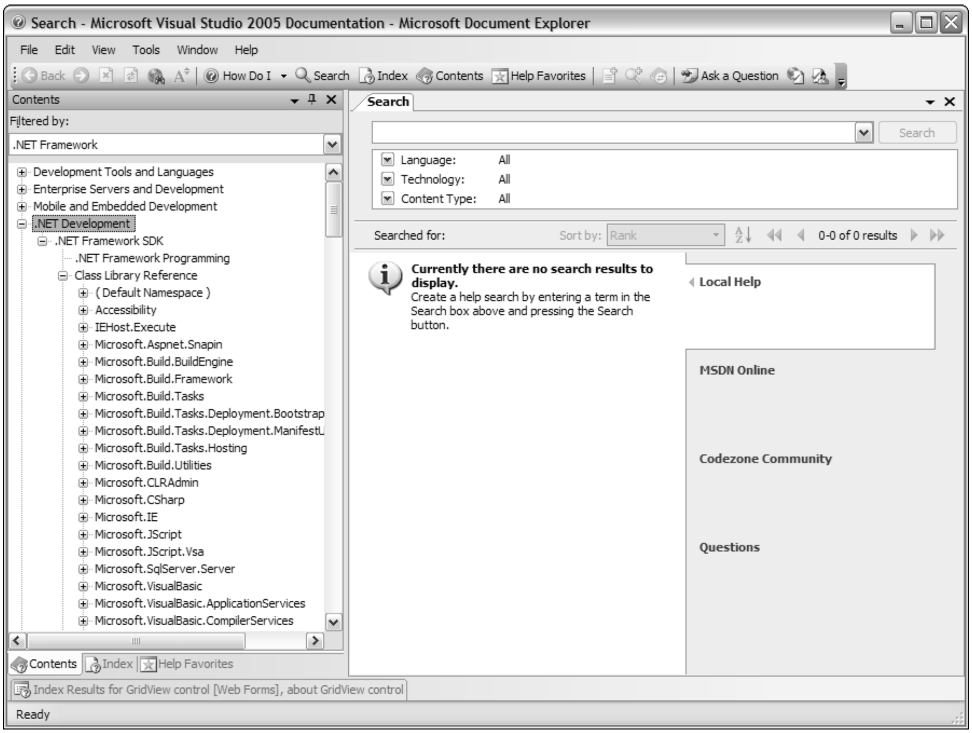


Figure 14-1

Another source of help on the .NET Framework 2.0 is the .NET Framework 2.0 SDK. At the time of writing it's available for downloading from `http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=FE6F2099-B7B4-4F47-A244-C96D69C35DEC`. If the download moves at a later date, try a Google search using ".NET Framework 2.0 SDK site:microsoft.com" to locate the download page.

You can also access information about .NET Framework 2.0 classes online at `http://msdn2.microsoft.com/en-us/library/aa139635.aspx`. URLs have been changing recently. If the preceding URL doesn't work, visit `http://msdn2.microsoft.com/en-us/netframework/default.aspx` and look for information on .NET 2.0.

There are an enormous number of .NET classes. If you are new to the .NET Framework, you should assume that you will have to invest a significant amount of time in order to get up to speed with the .NET Framework class library unless your .NET scripting needs are very focussed or you find a script that suits you in a script library. When beginning to learn the .NET Framework, I suggest that you start with a class that you might use frequently and really get to know it, as well as using your increasing knowledge of that class to understand how the help files of the class library are laid out.

Scripting libraries are an additional source of useful information on using the .NET Framework. Currently, Microsoft's Scripting Center for PowerShell is located at `www.microsoft.com/technet/scriptcenter/hubs/msh.mspx`.

Creating .NET Objects

The `new-object` cmdlet helps you create .NET objects and COM objects. It is described in the next section. However, although you will probably use it frequently to create a new .NET object, the `new-object` cmdlet is not the only way to create a new .NET object.

You can also create an object by casting a string to the desired object type, assuming that the cast is a valid one. I describe that technique in the section following the `new-object` cmdlet section.

The new-object Cmdlet

The `new-object` cmdlet is used to create new .NET objects and new COM objects. Using the `new-object` cmdlet to create COM objects is described in Chapter 13. In this section, I focus on using the `new-object` cmdlet to create new .NET objects.

The `new-object` cmdlet supports the following parameters (in addition to the common parameters, which are described in Chapter 6):

- `TypeName` — The fully qualified name of the .NET type to be created. It is both a required parameter and a positional parameter.
- `ArgumentList` — Arguments to the constructor of the type specified in the `Arguments` parameter. An optional parameter.
- `ComObject` — Used only when creating COM objects.
- Strict — Used only when creating COM objects.

The `-ComObject` and `-Strict` parameters relate specifically to the use of the `new-object` cmdlet with COM objects and is covered in Chapter 13.

To create a new .NET object, you simply provide a valid .NET type name as the value of the `TypeName` parameter to the `new-object` cmdlet. The value of the `TypeName` parameter specifies the .NET type of the object to be created. For example, to create a new `System.DateTime` object, use the following command:

```
$myDate = new-object -Typename System.DateTime
```

When you create an object whose class belongs to the `System` namespace, it isn't necessary for you to provide the full class name. However, I suggest you do that routinely, since it will be required when you create objects with classes in any other .NET Framework namespace.

Either of the following commands allows you to demonstrate that a new `System.DateTime` object has been created.

```
$myDate.GetType()
$myDate.GetType().Fullname
```

As you can see in Figure 14-2, these commands create a new `System.DateTime` object and assign it to the variable `$myDate`. The same technique can be used to create a new object of any other .NET type.

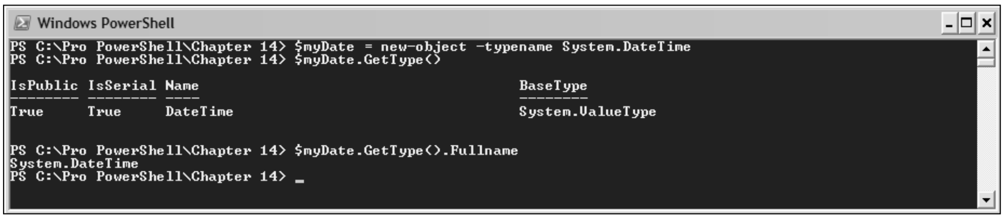


Figure 14-2

The `System.DateTime` object exists, represented by the `$myDate` variable, but you can't currently do much with it, since you didn't assign it a datetime value when you created the object. You can't, for example, manually set the values of properties such as `$myDate.Year` to a desired value, since the `Year` property and similar properties are read-only. You can inspect the properties, using a command like the following, which I filtered to show the `Year`, `Month`, and `Day` properties, among others.

```
$myDate |
get-member -memberType property |
where-object {$_.name -match "[ymd].*"}
```

Figure 14-3 shows the result of running the preceding command. Notice that each of the `Year`, `Month`, and `Day` properties (as well as some others) only have `get()` methods. In other words, you can't set their values.

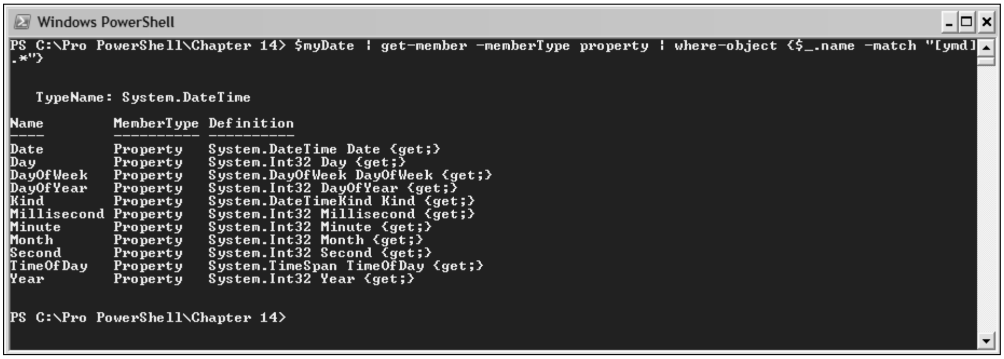


Figure 14-3

An alternative approach is to look at the available methods. The following command should display all method names beginning with g or s. That should display all get and set methods for the variable `$myDate`.

```
$myDate |
get-member -memberType method |
where-object {$_.name -match "[gs].*"}
```

Executing the command on a variable of type `System.DateTime` doesn't display any set methods, only get methods. In other words, the corresponding properties are read-only.

For some objects, you won't be allowed to create an object with only a `TypeName` parameter. The constructor(s) for the class can be expected to require one or more arguments. If you don't supply a value for the `-argumentList` parameter of the `new-object` cmdlet in PowerShell, then that is equivalent to attempting to using a constructor that is not available in .NET 2.0. For example, if you attempt to create a `String` object using this command:

```
$myString = new-object -typename System.String
```

and don't use the `-argumentList` parameter, an error message is displayed:

```
New-Object : Constructor not found. Cannot find an appropriate constructor for type
System.String.
At line:1 char:23
+ $myString = new-object  <<<< System.String
```

The constructors for a `System.String` object require 1, 2, 3, or 4 arguments. Omitting the `-argumentList` parameter causes the `new-object` cmdlet to attempt to use a constructor with 0 arguments, which is not supported.

In situations like these, you need to supply arguments when creating a new .NET object. For example, in earlier commands you created a `System.DateTime` object, but its important properties are read-only. So, you can't supply new values for them once you have created the object.

You supply arguments when creating a new object as a comma-separated list that is the value of the `-argumentList` parameter. The values you supply to the `-argumentList` parameter correspond to the values to be supplied in the constructor(s) for an object. Some .NET objects have multiple overloads. Each overload has a corresponding argument list as the value of the `-argumentList` parameter.

> *Many .NET classes support multiple constructors. The method is overloaded. It can have different numbers of arguments, which may of different types.*

To create a `System.DateTime` object representing a date of 2007/08/31, use either of the following commands. The parentheses are not necessary, but since constructors in other languages use parentheses I tend to use them in PowerShell, too.

```
$myDate2 = new-object -typename System.DateTime -argumentList (2007,08,31)
```

Or:

```
$myDate2 = new-object -typename System.DateTime -argumentList 2007,08,31
```

In the preceding code, under the covers, you use the constructor for `System.DateTime` that has three arguments. There are 11 available constructors for `System.DateTime`. Only one constructor takes three arguments. Thus, you supply three comma-separated values in the value of the `-argumentList` parameter. The arguments are `Int32` values.

There is also a constructor for `System.DateTime` that has six arguments. You can use this to create a `System.DateTime` object and specify year, month, day, hour, minute, and second values. Again, each of those is an `Int32` value. The following command creates a `System.DateTime` object for 20 minutes and 10 seconds past noon on 2007/08/31:

```
$myDate3 = new-object -typename System.DateTime -argumentList (2007,08,31,12,20,10)
```

You can demonstrate how each of the values in the argument list has been used by the constructor using the following commands:

```
$myDate3.Year
$myDate3.Month
$myDate3.Day
$myDate3.Hour
$myDate3.Minute
$myDate3.Second
```

Figure 14-4 shows the results of executing the preceding commands.

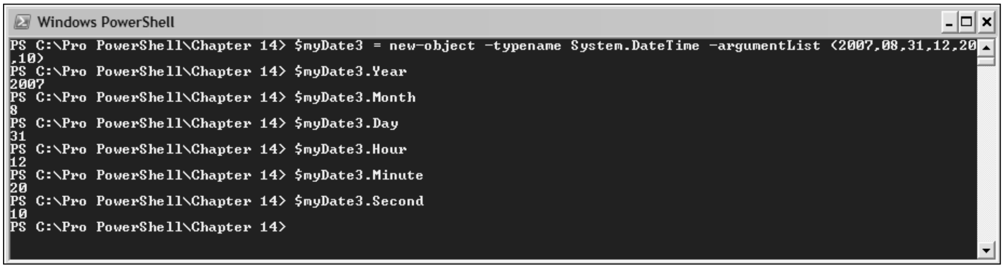


Figure 14-4

With some classes you cannot provide arguments where you might assume that they make sense. For example, if you want to create a `System.Boolean` object, you must first create the object and then, in a separate step, assign a Windows PowerShell `Boolean` value to it. The following command creates a `System.Boolean` object:

```
$myBoolean = new-object -Typename System.Boolean
```

By default, the `System.Boolean` object is created with the value `False`. To change it to `$true`, you need to execute the following statement:

```
$myBoolean = $true
```

Figure 14-5 also shows the behavior, including the error message received when trying to use the `Arguments` parameter for this object.

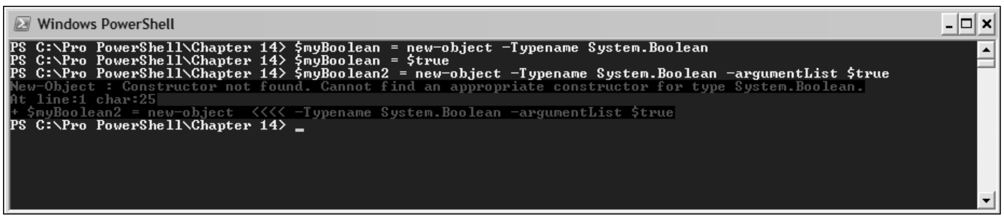


Figure 14-5

If you attempt to supply a value in the `-argumentList` parameter, you see the following error message:

```
New-Object : Constructor not found. Cannot find an appropriate constructor for type
System.Boolean.
At line:1 char:24
+ $myBoolean = new-object  <<<< -Typename System.Boolean -argumentList $true
```

Be careful when creating a `System.DateTime` object how you supply arguments to the `new-object` cmdlet. Figure 14-6 shows two forms of syntax that you might expect to work, which generate error messages.

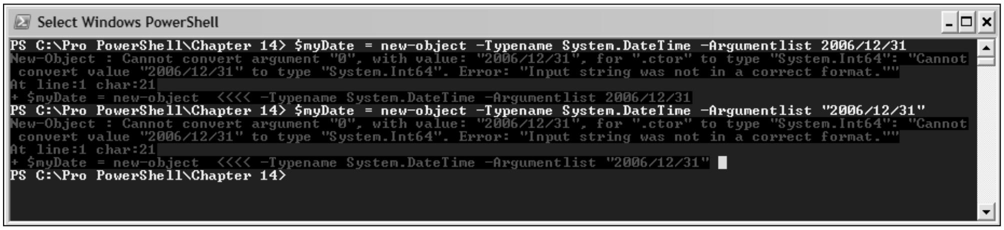


Figure 14-6

The following commands fail to work:

```
$myDate = new-object -Typename System.DateTime -Argumentlist 2006/12/31
$myDate = new-object -Typename System.DateTime -Argumentlist "2006/12/31"
```

The error message hints at the cause.

```
New-Object : Cannot convert argument "0", with value: "2006/12/31", for ".ctor" to
type "System.Int64": "Cannot
 convert value "2006/12/31" to type "System.Int64". Error: "Input string was not in
a correct format.""
At line:1 char:21
+ $myDate = new-object  <<<< -Typename System.DateTime -Argumentlist "2006/12/31"
```

The value you supplied for the `-argumentList` parameter is being interpreted as a single value, since there are no commas to indicate multiple values. The value is interpreted as an `Int64` value, which is the only constructor for a new `System.DateTime` with one argument.

Potentially more dangerous is when you seem to have successfully created an object but haven't achieved the desired setting of properties of interest. The following statement successfully creates a `System.DateTime` object for 2006/12/31:

```
$myDate = new-object -Typename System.DateTime -Argumentlist 2006,12,31
```

If you use either of the following commands (with one set of paired double quotation marks around the arguments or paired apostrophes), an object is created, but the value of properties such as `Year` are not set as you might expect, as shown in Figure 14-7. Notice that no error message is displayed.

```
$myDate = new-object -Typename System.DateTime -Argumentlist "2006,12,31"
```

Or:

```
$myDate = new-object -Typename System.DateTime -Argumentlist "2006,12,31"
```

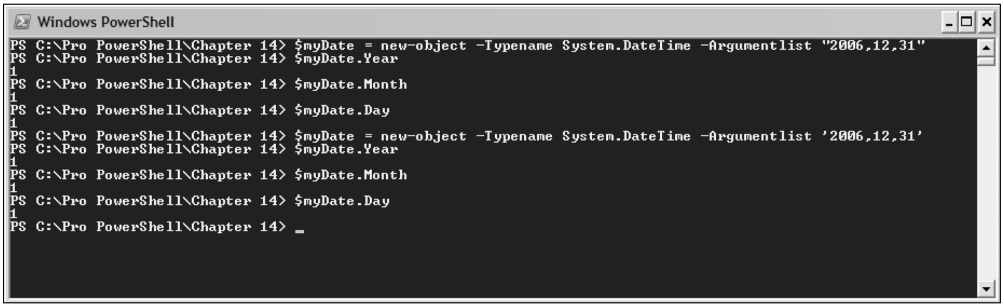


Figure 14-7

If you are going to use paired quotation marks or paired apostrophes in such a command, you must create a pair of double quotation marks or apostrophes for each argument in the list, as shown in the following commands:

```
$myDate = new-object -Typename System.DateTime -Argumentlist "2006","12","31"
$myDate = new-object -Typename System.DateTime -Argumentlist '2006','12','31'
```

When you construct the commands in that way, the object is created and its properties correctly set, as illustrated in Figure 14-8.

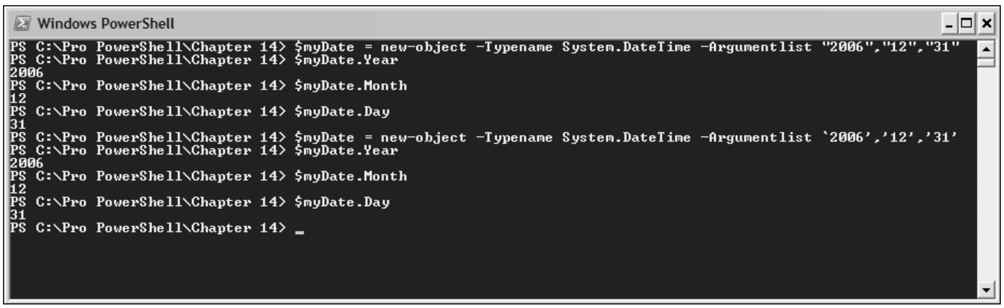


Figure 14-8

The `System.DateTime` constructor is more complex than many. As you have seen, it's important to be sure what is happening when you create a new object, even if no error message is displayed.

Other Techniques to Create New Objects

The `new-object` cmdlet is not the only way to create new .NET objects. In fact, the technique of creating new .NET objects that you will likely use most frequently is the implicit creation of one or more new .NET objects when you execute a statement or pipeline.

The following command assigns a string value to a variable and also creates a new object of type `System.String`:

```
$myString = "Hello world!"
```

If you type the following command, you can demonstrate that a `System.String` object has been created:

```
$myString.GetType().Fullname
```

Similarly, if you type the following command, you create several `System.Diagnostics.Process` objects held in an array:

```
$Processes = get-process
```

If you execute the following command, you can confirm the creation of a `System.Diagnostics.Process` object for each element in the array:

```
$Processes[0].GetType().Fullname
```

You can also cast a string variable or string literal to another .NET type. Casting (also called type casting or datatype conversion) is a way to convert an object of one datatype to an object of another datatype.

The value

```
"2006/12/31"
```

is a string. If you assign it to a variable, that variable is of type `System.String`. Figure 14-9 demonstrates this.

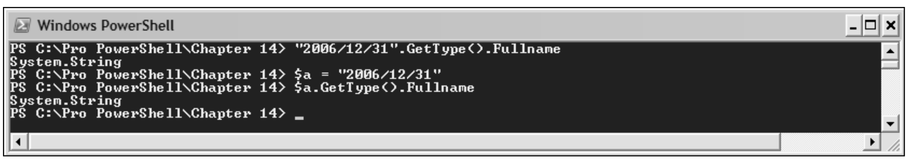


Figure 14-9

You can convert what is a string value to a `System.DateTime` object. You can either cast the string to a `System.DateTime` object and then assign that to a variable. Or you can assign a string value to a typed variable. To create the `$a` object using the first approach I just described, use this syntax:

```
$a = [System.DateTime]"2006,12,31"
```

First, the string value `2006,12,31` is cast to a `System.DateTime` object. You can demonstrate that cast using the following command:

```
([System.DateTime]"2006/12/31").GetType().Fullname
```

Alternatively, you can specify the type of the variable and then assign a string value to it:

```
[System.DateTime]$a = "2006/12/31"
```

The string value, if it is a value that can be cast to a `System.DateTime` value, is used to create a new `System.DateTime` object.

As shown in Figure 14-10, these techniques work to produce a new `System.DateTime` object like that produced using the `new-object` cmdlet, as described in the preceding section.

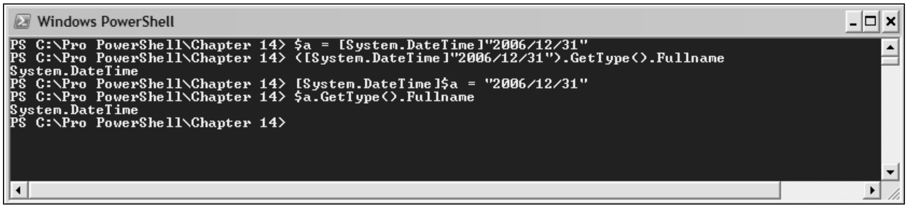


Figure 14-10

If the string can be cast to a datetime value, then the `System.DateTime` object is created. If the string can't be cast, for example:

```
$a = [System.DateTime]"Hello world!"
```

or:

```
[System.DateTime]$a = "Hello world!"
```

then the following error message is displayed:

```
Cannot convert value "Hello world!" to type "System.DateTime". Error: "The string
was not recognized
 DateTime. There is a unknown word starting at index 0."
At line:1 char:23
+ $a = [System.DateTime]" <<<< Hello world!"
```

However, the syntax you use when employing this technique is different from that you use with `new-object`. For example, as shown in Figure 14-10, the following command:

```
$a = [System.DateTime]"2006/12/31"
```

successfully creates a `System.DateTime` object but, as you may recall, the command

```
$myDate = new-object -Typename System.DateTime -Argumentlist "2006/12/31"
```

produced an error. The difference is that, when you use a cast where the value between paired double quotation marks is a `String` value, that value is interpreted as a date, whereas the `new-object` syntax was attempting to interpret `"2006/12/31"` as a single argument to be parsed as a an `Int64` value.

Inspecting Properties and Methods

Once you have created a new .NET object, you will frequently want to inspect or change properties of that object or use its methods to carry out specific tasks. One of the major learning tasks if you are new to the .NET Framework is to become familiar with the members of the huge variety of .NET classes. To help you explore, PowerShell provides a cmdlet, `get-member`, to assist you in finding the members of any .NET class that you need to use.

Using the get-member Cmdlet

The `get-member` cmdlet is very useful for finding the members of a .NET instance object. In addition to the common parameters, it supports the following parameters:

- `Name` — Specifies what member names are to be selected. A positional parameter in position 1. The default value is the wildcard *, which matches all members.
- `InputObject` — Specifies what object or objects are the input to the cmdlet. Used when the `get-member` cmdlet is not receiving input objects from a pipeline.
- `MemberType` — Specifies the type of member to be returned. Allowed types are enumerated later in this section.
- `Static` — Specifies that only static members are to be returned

One way to explore the members of the `$myDate` object is the command

```
$myDate |
get-member
```

which pipes the `$myDate` object to the `get-member` cmdlet. This is equivalent to, but simpler than, the following command:

```
$myDate|
get-member -Name * -MemberType All
```

The preceding command returns about two screens of members of the `$myDate` object, depending on how you have the Windows PowerShell window sized. If you prefer, you can page the output using the command:

```
$myDate |
get-member |
more
```

or:

```
$myDate |
get-member |
out-host -paging
```

Often you will want to filter the members in some way. For example, to find only members that are methods, use this command:

```
$myDate |
get-member  -MemberType method
```

To find only properties, use the following command:

```
$myDate |
get-member -MemberType Property
```

By default, members of an object may not be returned in alphabetical order. To achieve alphabetical order, use the `sort-object` cmdlet in the pipeline, as shown here:

```
$myDate |
get-member -MemberType Property |
sort-object Name
```

Figure 14-11 shows the results of executing the preceding command.

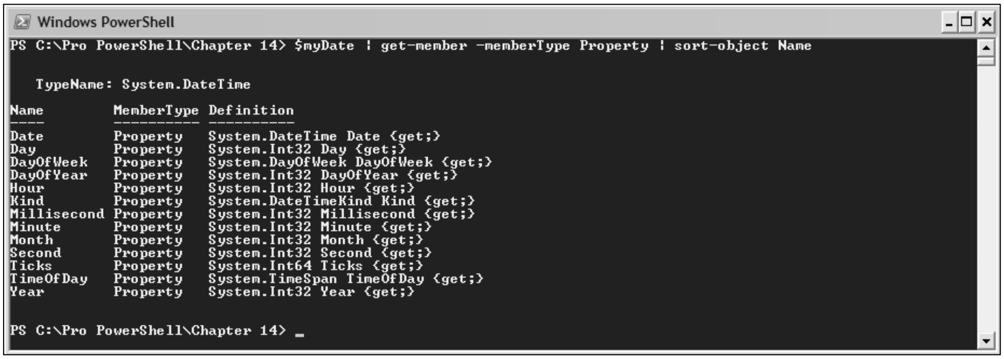


Figure 14-11

Notice that all of the properties displayed in Figure 14-11 only show get methods. This signifies that the properties are read-only.

However, on other objects the properties may be read-write. For example, on a `System.IO.FileInfo` object several of the properties are read-write, as you can demonstrate by executing the following command:

```
get-childitem |
get-member -memberType Property
```

Figure 14-12 shows the results of executing the preceding command. Notice that several properties have both get and set methods indicated.

Figure 14-12

As mentioned earlier in the section the members just described exist on instance objects. There are also *static* methods that exist on the .NET classes from which instance objects are derived. To retrieve a list of static methods for a .NET class, sorted by the value of the `Name` property, use the following command:

```
$myDate |
get-member -MemberType method -Static |
sort-object Name
```

The presence of a static method means that you can use the method without having to create an instance object. Figure 14-13 shows the results of executing the preceding command. Notice that the word `static` appears in the Definition column for each of the static methods listed in Figure 14-13.

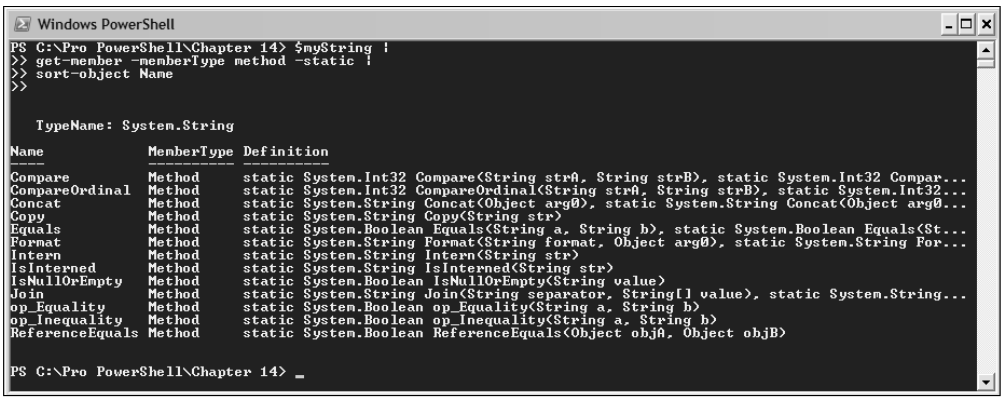


Figure 14-13

Modifying the value of the `MemberType` parameter allows you to selectively retrieve information about other types of members.

When members are displayed as determined by the default formatter a tabular layout is produced. The `Definition` property is included in the display, but it may not be possible to display the full definition of an object, as you can see with some of the definitions in Figure 14-13.

Adding the `format-list` cmdlet in the final step of a pipeline can help you see the full definition of a member. Careful inspection of the definition in the lower part of the figure shows two overloads for the `TryParseExact()` method with the return value specified before the method name, and the arguments and their types listed inside paired parentheses. This information allows you to correctly construct arguments for the methods without having to consult the documentation mentioned earlier in this chapter. Admittedly, the formal documentation is easier, but if you can understand the information that is at hand from the PowerShell command line it can save you quite a bit of time.

Compare the display produced about the `TryParseExact()` method, using this command:

```
$myDate |
get-member -Name TryParseExact -MemberType method -Static |
sort-object Name
```

and the output produced by:

```
$myDate |
get-member -Name TryParseExact -MemberType method -Static |
sort-object Name |
format-list
```

Figure 14-14 shows the two results.

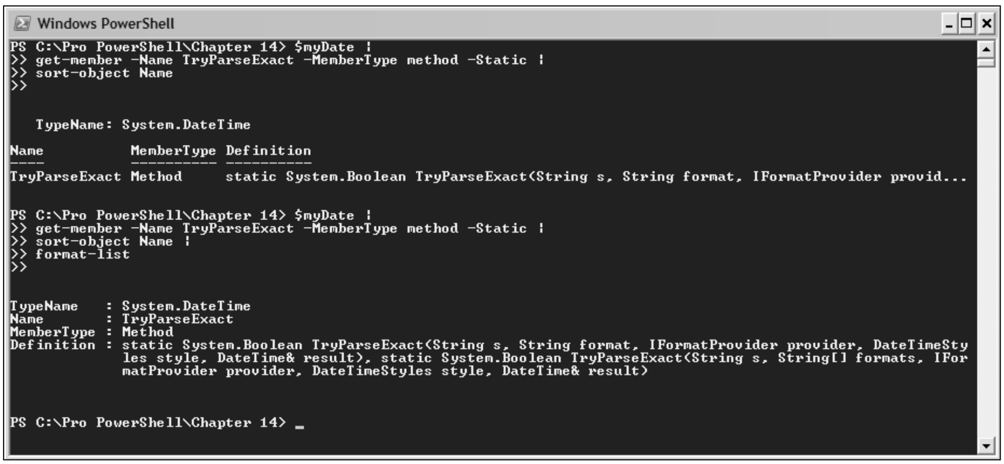


Figure 14-14

The following are the allowed values for the `-memberType` parameter: `AliasProperty`, `CodeProperty`, `Property`, `NoteProperty`, `ScriptProperty`, `Properties`, `PropertySet`, `Method`, `CodeMethod`, `ScriptMethod`, `Methods`, `ParameterizedProperty`, `MemberSet`, and `All`.

Using .NET Reflection

When you use Windows PowerShell to work with .NET Framework objects, you can take advantage of the functionality of .NET reflection. Reflection allows you to look inside an assembly and find out its characteristics. Inside a .NET assembly information is stored that describes what the assembly contains. This is called *metadata*. A .NET assembly is, in a sense, self-describing, at least if interrogated correctly.

In the .NET Framework, assemblies contain modules, modules contain classes, and classes contain members. Reflection allows you to explore the hierarchy of assemblies, modules, classes, and members.

The `System.Reflection` namespace, on which .NET reflection is based, is an extensive one. A full discussion of reflection is beyond the scope of this chapter, but the following introduction should help you get started so that you can, from Windows PowerShell, explore in detail the characteristics of .NET classes and types that interest you.

Reflection uses the members of the `System.Reflection` namespace together with `System.Type`. In .NET a namespace is simply a named collection.

In the following sections, I introduce several methods that you can use to find detailed information about members, methods, and properties of a .NET class. As the following sections make clear, the method you use will depend on your existing knowledge about the class. These methods can be combined with use of the `get-member` cmdlet and the `GetType()` method, if you are starting from an instance object and want to find out more information about its class.

Using the GetMembers() Method

The `GetMembers()` method returns an array of `MemberInfo` objects. Depending on permissions, the method either returns all public members of a class or all members of the class. You use it with a .NET type, not with an instance object. The `GetMembers()` method is used with the following general syntax:

```
[DotNetType].GetMembers()
```

The name of the .NET type must be enclosed in paired square brackets. The type name in square brackets is associated with the method using dot notation.

For example, to find the members of the `System.DateTime` class (as opposed to the members of an instance object), use the following command:

```
[System.DateTime].GetMembers()
```

Figure 14-15 shows one screen of the results returned by the `GetMembers()` method. This shows information about the `get_Hour()` method. Notice its `Name`, `get_Hour`, its `MemberType`, `Method`, its `Return Type`, `System.Int32`, that it's a public method (the value of `IsPublic` is `True`).

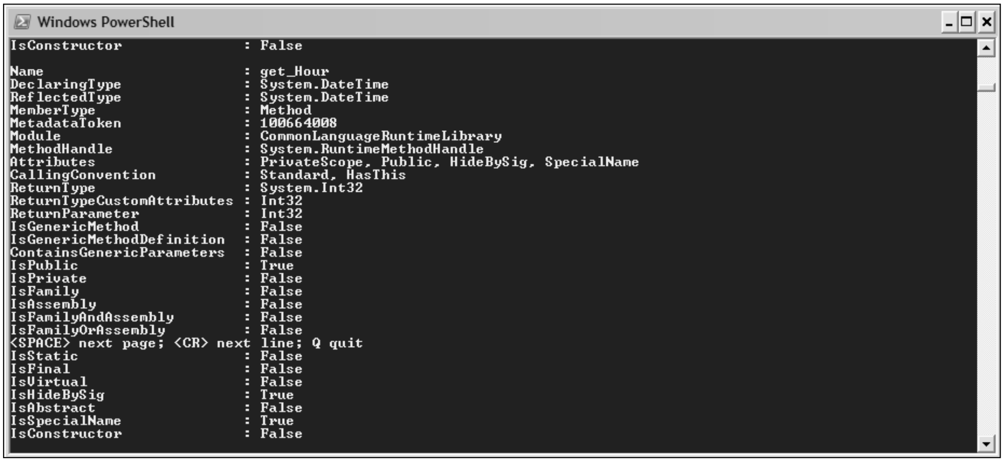


Figure 14-15

If you want information about the `GetMembers()` method itself, simply omit the paired parentheses from the command:

```
[System.DateTime].GetMembers
```

As shown in Figure 14-16, information about the characteristics of the `GetMembers()` method is displayed.

Figure 14-16

Notice that two overloads are listed in the `Overload Definitions` property. The second definition:

```
System.Reflection.MemberInfo[] GetMembers()
```

is the one we used earlier. It takes no arguments and returns an array of `System.Reflection.MemberInfo` objects.

Using the GetMember() Method

The `GetMember()` method allows you to get information about a specified member of a class. It returns an array of `MemberInfo` objects. If you know a class well, you may go directly to a command such as

```
[System.DateTime].GetMember("Year")
```

which displays information about the `Year` property of the `System.DateTime` class. Figure 14-17 shows the result of executing the preceding command. You can see that the name of the member is `Year`, that it is a property, that its type is `System.Int32`, and that you can read it but not write it. In other words, it's a read-only property.

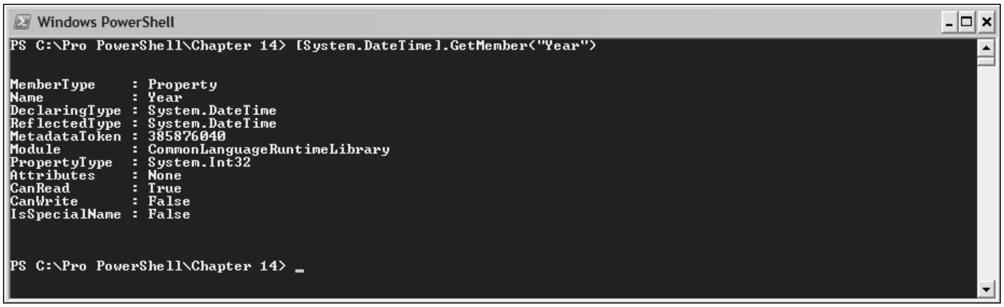


Figure 14-17

If you have limited knowledge of the class, you may precede that command with a command such as

```
[System.DateTime].GetMembers()
```

to find out what members the class has, as explained in the preceding section.

The `GetMember()` method can also be used with wildcards to return information on multiple members. For example, to return information on all members of the `System.DateTime` class beginning with the letter D, use this command:

```
[System.DateTime].GetMember("D*")
```

Since you are working inside the .NET Framework, it is better to assume that the names of members are case-sensitive. The information you see will often, but not always, depends on the case you use. As you can see at the top of Figure 14-18, using the command

```
[System.DateTime].GetMember("d*")
```

returns nothing, since the names of the members have an initial uppercase letter.

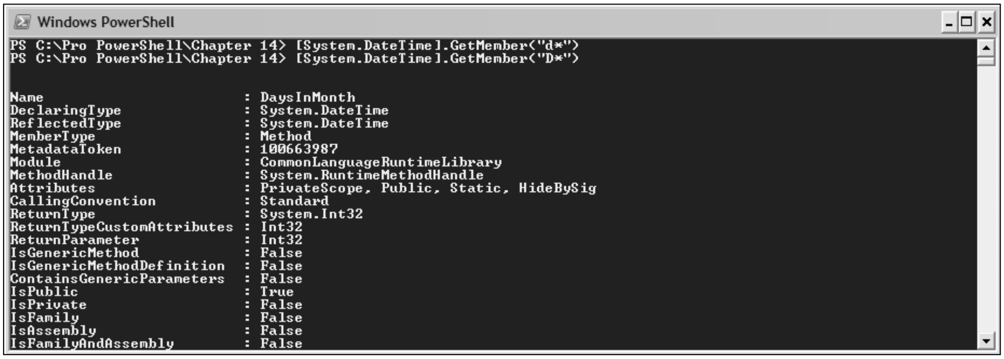


Figure 14-18

In other situations, case is important, and depending on what case, you use you will retrieve and display a different set of objects. Compare the results in Figure 14-19 of executing the following two commands (the first using `g*` and the second using `G*` as the wildcard to identify member names):

```
[System.DateTime].GetMember("g*") |
format-list Name
```

Compare it to the results produced by the following command:

```
[System.DateTime].GetMember("G*") |
format-list Name
```

However, case may not matter. For example, although when you use the `GetMember()` method, you must use a lowercase `g` to retrieve information about the `get_Day()` method of the `System.DateTime` class, you can use either uppercase or lowercase initial letter when you use the method in PowerShell, as you can confirm by executing both the following commands:

```
$myDate.get_Day()
```

and:

```
$myDate.Get_Day()
```

Generally, if you are working outside PowerShell, I suggest that you assume that case-sensitivity is operative. Inside PowerShell you can generally assume that case-insensitivity is operative.

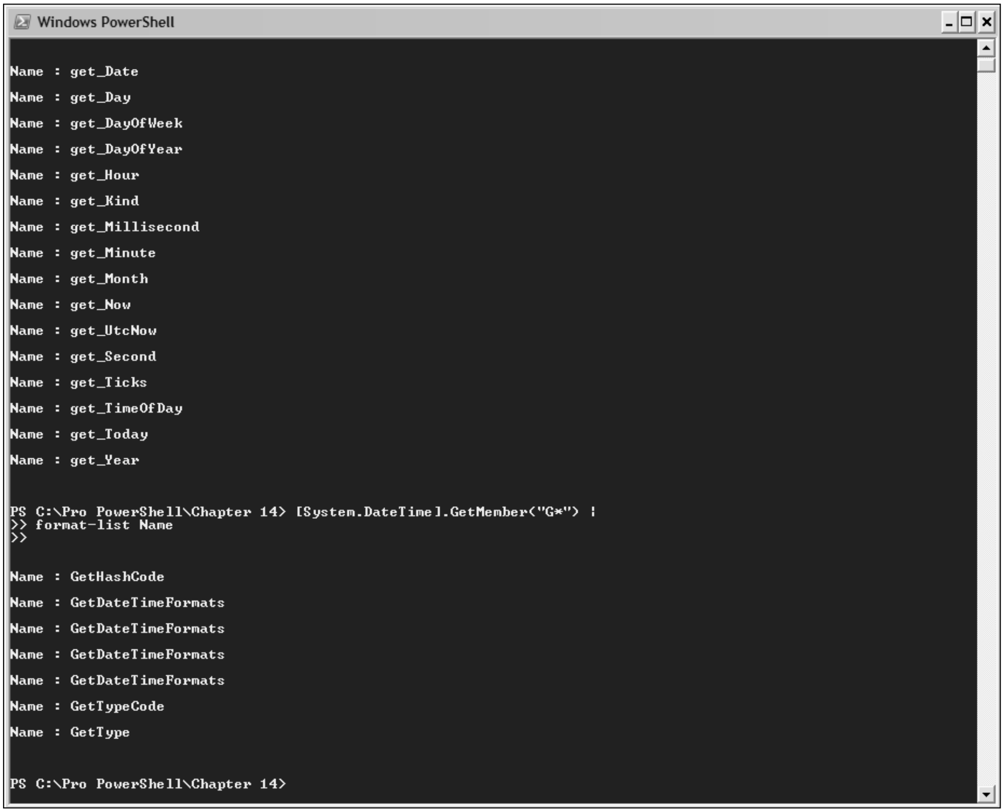


Figure 14-19

Using the GetMethods() Method

The `GetMethods()` method retrieves an array of `MethodInfo` objects. Just as you can use the `get-member` cmdlet with the `MemberType` property set to `Method` to display the methods of an instance object using a command like:

```
$myDate |
get-member -memberType Method
```

so you can use the `GetMethods()` method of a class to retrieve only methods of a class. The following command displays one screen of the method of the `System.String` class:

```
[System.String].GetMethods() |
more
```

The `GetMethods()` method displays much more information about a method than the `get-member` usage mentioned above. Figure 14-20 illustrates this.

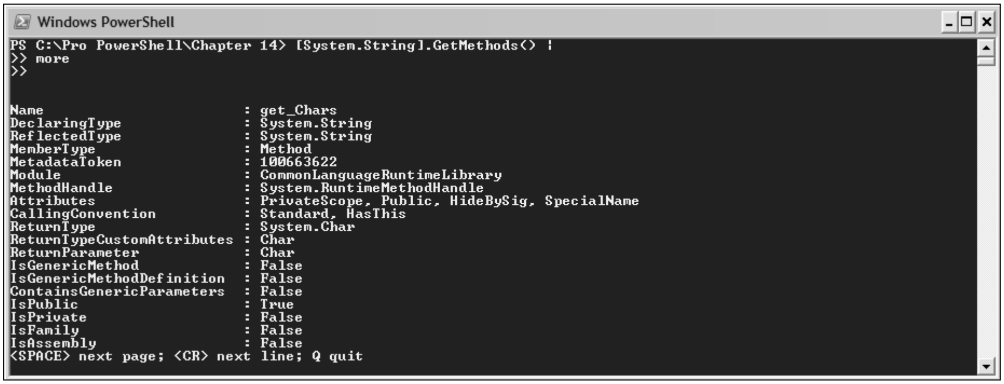


Figure 14-20

Using the GetMethod() Method

The `GetMethod()` method retrieves an array of `MethodInfo` objects. When you use the `GetMethod()` method, you supply the name(s) of methods of interest. The `GetMethods()` method in the usage described in the preceding section takes no argument and displays all methods of a class. Often you will supply the name of a method as a string literal. Equally, as demonstrated in Figure 14-21, you can supply the argument to the `GetMethod()` method as a variable.

```
$a = "GetType"
[System.DateTime].GetMethod($a)
```

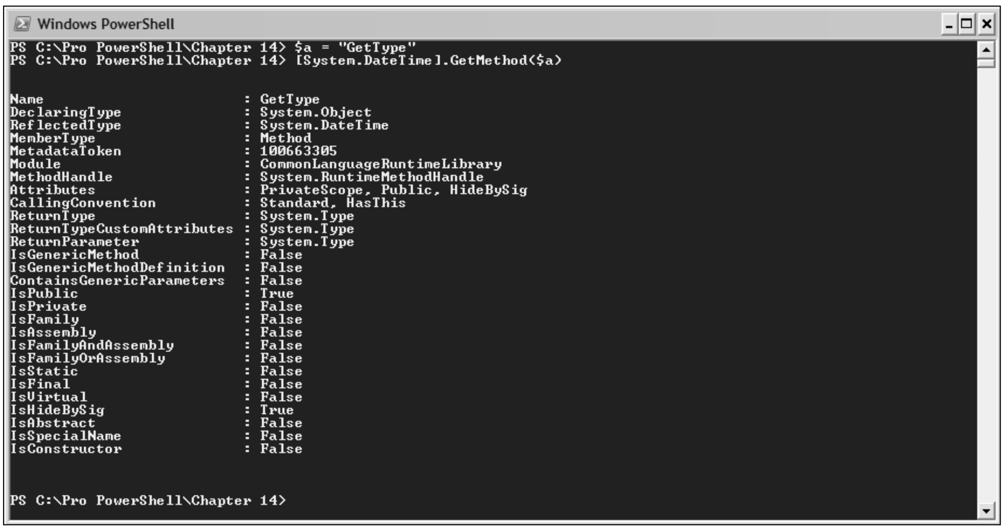


Figure 14-21

The first argument to the `GetMethod()` method can also include a wildcard.

Using the GetProperties() Method

The `GetProperties()` method retrieves an array of `PropertyInfo` objects. Using the `GetProperties()` method is broadly similar to using the `get-member` cmdlet with the value of the `memberType` parameter set to `Property`, but it returns the properties of a class.

To produce an easily read list of the properties of the `System.DateTime` class, you can use the `sort-object` and `format-list` cmdlets to display an alphabetical list of the names of the properties:

```
[System.DateTime].GetProperties() |
sort-object Name |
format-list Name
```

As you can see in Figure 14-22, this provides a more convenient way to explore the properties of a class than displaying multiple screens of information. Once you find a desired property, use the `GetProperty()` method described in the next section to display full information for that property.

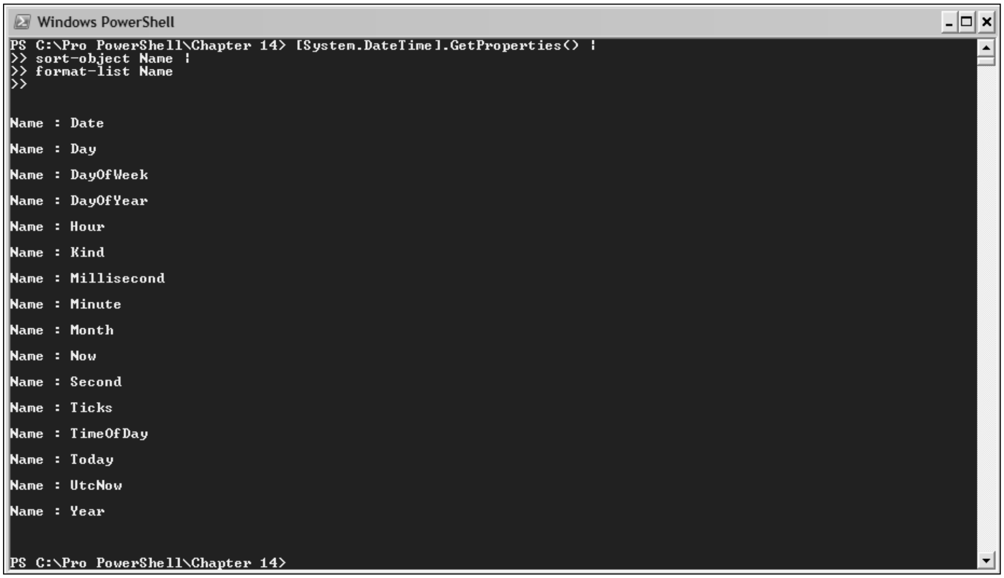


Figure 14-22

You can use the `GetProperties()` method together with the `where-object` cmdlet to find all read-only properties of a type. For example, to find the read-only properties of the `System.DateTime` type, use the following code:

```
[System.DateTime].GetProperties() |
where-object {$_.CanWrite -eq $False} |
where-object {$_.CanRead -eq $True}
```

Figure 14-23 shows the first two read-only properties of the `System.DateTime` class.

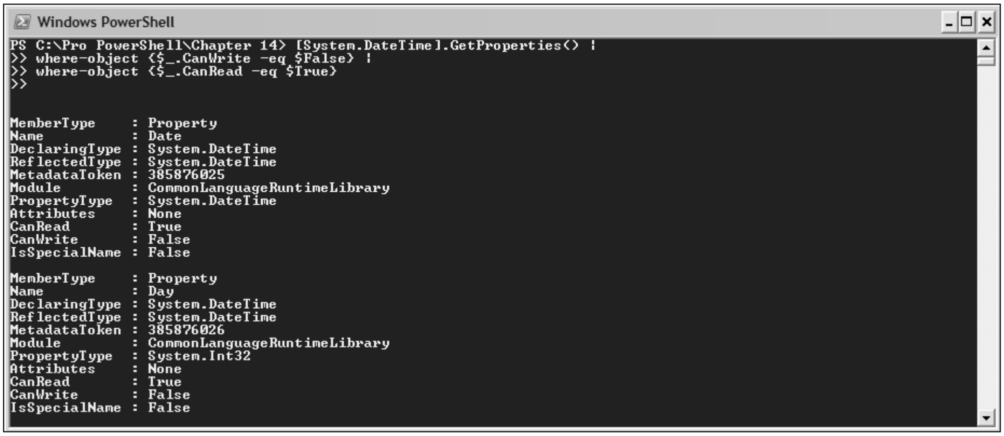


Figure 14-23

Using the GetProperty() Method

The `GetProperty()` method returns an array of `PropertyInfo` objects corresponding to the property or properties whose name is the argument to the `GetProperty()` method. You can use wildcards in the first argument to the `GetProperty()` method.

For example, to find out the characteristics of the `Year` property of the `System.DateTime` class, use the following command:

```
[System.DateTime].GetProperty("Year")
```

or:

```
[System.DateTime].GetProperty('Year')
```

As you can see in Figure 14-24, the `Year` property is read-only. You can deduce that since the value of `CanRead` is `True` and the value of `CanWrite` is `False`.

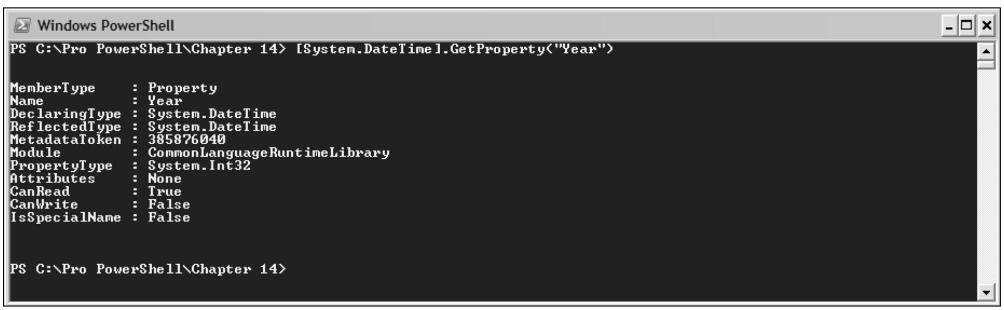


Figure 14-24

Using the `GetProperty()` method as shown in the preceding code is the simplest way to use the `GetProperty()` method. It can also be used with the form:

```
[DotNetClass].GetProperty(PropertyName, BindingFlag)
```

There are about 20 binding flags available to you. These binding flags affect how members and types are conducted by reflection. The binding flags are enumerated in the documentation of the `BindingFlags` enumeration. Using:

```
[System.DateTime].GetProperty("Year")
```

is equivalent to using:

```
[System.DateTime].GetProperty("Year", Default)
```

but the latter syntax seems not to work in the initial release of PowerShell 1.0. The `Default` binding flag indicates that no binding flags are set. The `BindingFlags` enumeration is used widely to control binding for classes in the `System.Reflection` namespace. Among the methods that use the `BindingFlags` enumeration are `GetMembers()`, `GetMember()`, `GetProperty()`, `GetProperties()`, `GetMethod()`, and `GetMethods()`.

You may have an object whose members you want to find. The following approach illustrates how you might go about it using a simple `String` object. For some classes that you use frequently, the pieces of information sought will become second nature before long, but when you first begin working with Windows PowerShell and .NET classes this is a useful way to characterize an object or class.

```
$myString = "Hello world!"
$myString | get-member -memberType Property
$myString.GetType()
[System.String].GetProperty("Length")
```

After creating the `$myString` variable in the first line, you can use the `get-member` cmdlet as the second step of the pipeline to obtain all the properties of the string object. The `memberType` parameter specifies that you only want information about properties to be returned. The third command retrieves information about the full name of the type of the string object represented by `$myString`. To obtain the full .NET namespace name for this type, type:

```
$myString.GetType().Fullname
```

The final line of the code above uses the `GetProperty()` method to retrieve information about the `Length` property of the `System.String` type. Figure 14-25 shows the results of running these statements.

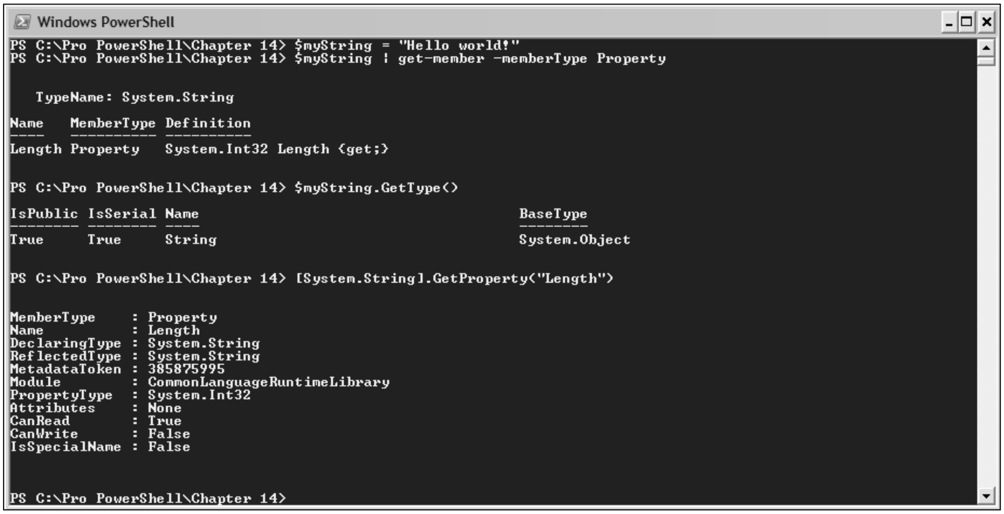


Figure 14-25

Using System.Type Members

The `System.Type` class — which is a basis of reflection, of course — also has members. You can use those to find out characteristics of the type.

For example, suppose that you want to find out what assembly a type is loaded from. You can use the following command to attempt to do that:

```
[System.String].Assembly
```

If the output from the preceding command is truncated on your system, use the `format-list` cmdlet to ensure that you can see all the attributes of the `Assembly` property:

```
[System.String].Assembly |
format-list.
```

Summary

Windows PowerShell is based on .NET objects. You can create new .NET objects using the following techniques:

- Using the `new-object` cmdlet
- Implicitly using a cmdlet or pipeline
- By explicitly casting a value to another compatible type

The `get-member` cmdlet allows you to explore the members of .NET instance objects, such as variables you use in your PowerShell commands.

You can also explore the members of .NET classes by using the `GetMembers()`, `GetMember()`, `GetMethods()`, `GetMethod()`, `GetProperties()`, and `GetProperty()` methods.

Part II

Putting Windows PowerShell to Work

Chapter 15: Using Windows PowerShell Tools for Discovery

Chapter 16: Security

Chapter 17: Working with Errors and Exceptions

Chapter 18: Debugging

Chapter 19: Working with the File System

Chapter 20: Working with the Registry

Chapter 21: Working with Environment Variables

15

Using Windows PowerShell Tools for Discovery

One of PowerShell's goals is to enable system administrators to find out what is happening on one system or, more usefully, on large numbers of systems. In this chapter, I explore techniques that allow you to use Windows PowerShell to find out what is happening on a system.

Of course, PowerShell isn't the only tool available to explore a Windows system. There are many other tools available from Microsoft and elsewhere that allow you to explore at least some aspects of what a system is doing. PowerShell is one tool in an armory, not the whole arsenal. But it's a very useful tool for exploration, partly for what it can do, partly for the convenient and interactive way that you can explore a system from the command line.

For example, if a machine has been showing sluggish performance, you might want to know what processes are running, how much CPU time they are using, and how much memory they are using. You can find out much of that information from Windows Task Manager, but it can be tedious either to scroll around within that utility's pretty constricted interface to change the columns to be displayed, or to scroll around an uncomfortably large number of columns to see the information you want. PowerShell lets you select and filter the information you want, display the parts of it that interest you onscreen and, if you want, send exactly the information you chose for *your* purposes to a file.

Similarly, you might want to know about services registered on a machine. Are the expected services running, for example? Or you might want to be able from the command line to stop and start a service.

Exploring System State

The Windows PowerShell shell maintains information about your system's current state. The information that the Windows PowerShell shell maintains about system state is summarized in the following table.

Information	Description
Current Working Location	The default location used by commands such as `get-childitem` if no path is explicitly provided
Error handling	Defines how errors are to be handled
Namespaces	Containers for names that ensure that any fully qualified name is unambiguous
Shell aliases	The aliases created by default when you start the Windows PowerShell shell
Shell functions	The functions created by default when you start the Windows PowerShell shell
Shell variables	The variables created by default when you start the Windows PowerShell shell

The sections that follow describe how you can access information about the system state.

Using the get-location Cmdlet

The `get-location` cmdlet lets you find the current working location in the context of a specified Windows PowerShell provider, which is supplied explicitly or by default. The `get-location` cmdlet supports the common parameters (covered in Chapter 6) and the parameters in the following list:

- `PSProvider` — specifies which Windows PowerShell provider to query. The default value is the current working provider.
- `PSDrive` — specifies a Windows PowerShell drive to query
- `Stack` — When specified displays the items on the current stack.
- `StackName` — Specifies the name of a stack for which the locations on the stack are to be displayed.

All of the preceding parameters are optional. The `get-location` cmdlet can be used to return `PathInfo` or `StackInfo` objects, depending on if you are retrieving locations from a PowerShell provider or items from a stack. When you intend the `get-location` cmdlet to return `PathInfo` objects, you either omit all parameters or use the optional `PSprovider` and `PSDrive` parameters. When you intend the `get-location` to return `StackInfo` objects, use the optional `Stack` and `StackName` parameters.

Not all combinations of the `Provider`, `Drive`, `Stack`, *and* `StackName` *parameters are valid You can use the* `PSProvider` *and* `PSDrive` *parameters together or you can use the* `Stack` *and* `StackName` *parameters together.*

The simplest use of the `get-location` cmdlet is this:

```
get-location
```

which finds the current working location in the current provider. If you are currently using the `FileSystem` provider, the preceding command will return the same result as:

```
get-location -PSProvider FileSystem
```

and:

```
get-location -PSProvider FileSystem -PSDrive C
```

> *If you are uncertain which is the current provider, use the* `get-childitem` *command and inspect the first line of output. If you are using the* `FileSystem` *provider, the first line will be similar to* `Directory: Microsoft.PowerShell.Core\FileSystem::C:\Documents` *and* `Settings\Andrew Watt`*, depending on the current working directory.*

Figure 15-1 shows the three commands in use when the current provider is the `FileSystem` provider.

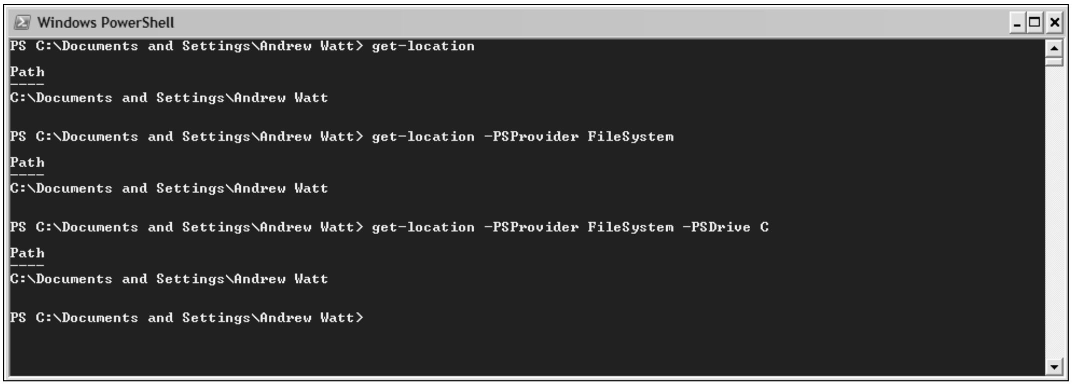


Figure 15-1

When you specify a value for the `Drive` parameter, you must omit the colon character. If you include the colon character, as here:

```
get-location -PSProvider FileSystem -PSDrive C:
```

you can expect the following error message to be displayed:

```
Get-Location : Cannot find drive. A drive with name 'C:' does not exist.
At line:1 char:13
+ get-location  <<<< -PSProvider FileSystem -PSDrive C:
```

Typically, you will know, for example, which provider you are using because of the name of the drive you are using, but if you want to display fuller information about the current location, you can create a simple pipeline and use the `format-list` cmdlet, as shown here:

```
get-location | format-list
```

Figure 15-2 shows the information displayed. Notice that information about the provider is included.

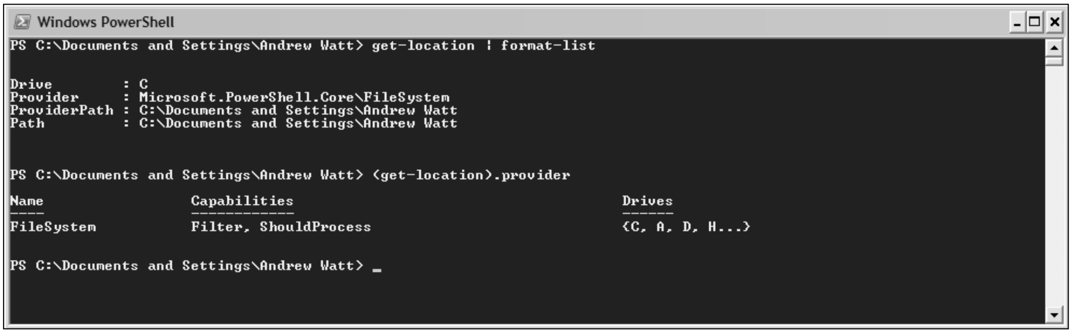


Figure 15-2

Alternatively, you can use the following command to display information about the current provider:

```
(get-location).provider
```

The term "current location" is arguably misleading, since you can have a "current location" on multiple drives at the same time. To view the current location for more than one provider, use the following command:

```
get-psdrive
```

Notice in Figure 15-3 that the current location in all drives where you have navigated away from the root of the drive is displayed.

Figure 15-3

When multiple current locations are displayed, you need to know which you are currently located in.

Current location does, of course, apply to your current location, but it also applies to the most recently accessed location on other PowerShell drives. As you saw in Figure 15-3, the current location on the `HKLM: drive` is `Software`. So, if you type the command:

```
set-location HKLM:
```

or use an alias:

```
cd HKLM:
```

you are taken to the most recently used location on that drive, in this case `HKLM:\Software`.

Using the `get-location` cmdlet with the `Stack` and `StackName` parameters is associated with use of the `push-location` and `pop-location` cmdlets, which I will describe now.

When you are repeatedly moving around a complex directory structure in the file system in Windows Explorer, you can use the Back and Forward buttons to move through the relevant folders. The stack of locations in PowerShell allows you to do something similar from the command line.

The push-location Cmdlet

The `push-location` cmdlet pushes a location on to the stack, which is a last in, first out data structure. In addition to supporting the common parameters, the `push-location` cmdlet supports the parameters in the following list:

- `Path` — The current working location is changed to the path specified using this parameter. Unlike the `-literalPath` parameter, this parameter accepts wildcards.
- `LiteralPath` — A literal value of the current working location to change to. This parameter does not accept wildcards.
- `StackName` — Specifies the stack to which the current working location or `PathInfo` object is pushed.
- `PassThru` — Passes the resulting object along the pipeline.

To push the current location on to the stack and change the current working directory to `C:\Windows`, use the following command:

```
push-location -Path C:\Windows
```

Once you have pushed a location on to the stack you can use the `get-location` cmdlet with the `Stack` parameter. The following command displays the locations pushed on to the stack, in this case a single location: `C:\Documents and Settings\Andrew Watt`.

```
get-location -Stack
```

You can use the command

```
pop-location
```

to return to the previous current working directory.

Figure 15-4 shows the results of executing the three preceding commands.

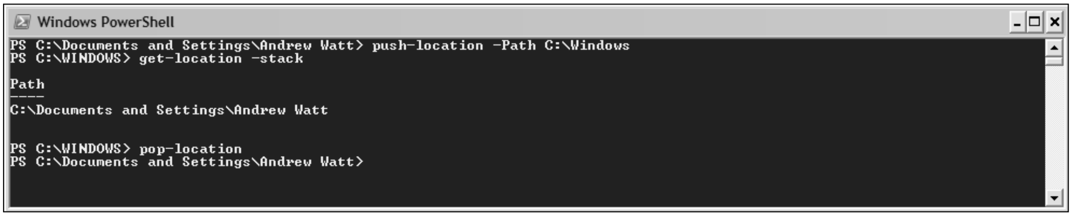


Figure 15-4

Using the `push-location` and `get-location` cmdlets becomes more helpful the more complex the path name (and the more folders you want to move between). If, for example, I navigate down a lengthy folder hierarchy in PowerShell to get to a folder like `C:\Program Files\Microsoft SQL Server\90\Samples\Integration Services\Package Samples\DataCleaning Sample`, I tend to do so in multiple steps. If I want to work in some other directory and return again to the DataCleaning Sample folder I can simply push it on to the stack, using

```
push-location -LiteralPath C:\Windows
```

do whatever I want in the chosen location and then use

```
pop-location
```

to return to the previous folder, as shown in Figure 15-5.

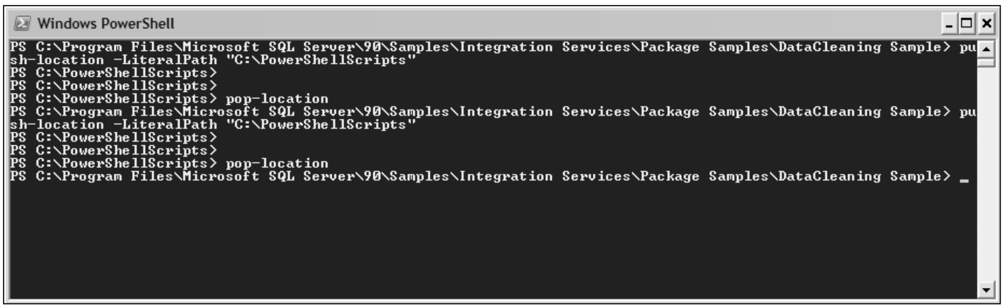


Figure 15-5

Another option to navigate the folders is to use the up arrow key in PowerShell to run through a series of commands like

```
set-location "C:\Program Files"
dir Mi*
set-location "Microsoft SQL Server"
dir
set-location 90
```

and so on and until I eventually get to the folder for the data cleaning sample. Using `pop-location` and `push-location` is simpler and is more useful if I want to navigate repeatedly between the two folders.

You can also use the `push-location` cmdlet to push a location on to a named stack. For example, suppose that you wanted to work between PowerShell, SQL Server SMO (SQL Server Management Objects), and Visual Studio 2005 project folders; you might create a named stack called PSSMO. You could work between the folders as just described for the default stack, but pushing locations onto a named stack means that it's there when you want to go back to that task.

If you want a named stack to be available every time you run PowerShell, add the relevant `push-location` *commands to the profile file to be executed at PowerShell startup.*

The following command pushes the current location, `C:\Program Files\Microsoft SQL Server\90\Samples\Engine\Programmability\SMO`, on to a stack named `PSSMO`.

```
push-location -StackName PSSMO -Path "C:\PowerShellScripts"
```

If you want, say, to navigate to a Visual Studio folder, too, you might add the command:

```
push-location -Stackname PSSMO -Path "C:\Documents and Settings\Andrew Watt\My
Documents\Visual Studio 2005\Projects"
```

Once you have pushed the locations on to a named stack, you can use the `get-location` cmdlet with the `StackName` parameter to retrieve the locations on that named stack. To retrieve the locations, in this case a single location, from the named stack `Fred`, use this command:

```
get-location -Stack -StackName PSSMO
```

This retrieves the single location just pushed to the named stack. However, the default stack can contain a different set of locations (which depends on how you have been using the `push-location` and `pop-location` cmdlets recently), as you can demonstrate by using this command:

```
get-location -Stack
```

Since there is no `StackName` parameter in the preceding command, the default stack is used.

Figure 15-6 shows the three preceding commands being used.

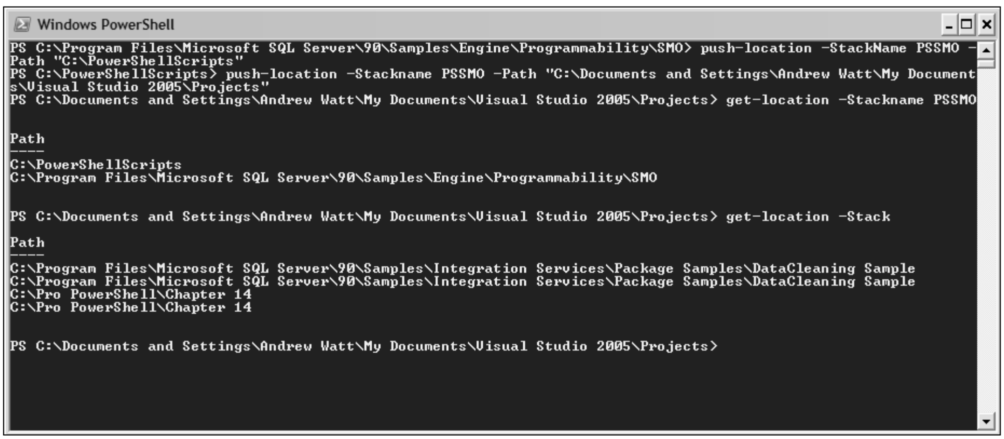


Figure 15-6

The PSSMO stack stays there in whatever state you leave it until you are ready to use it again. When you want to go back to work with, say, SMO and PowerShell again, you just pop a location from the stack and away you go.

Using the `-passThru` parameter with the `push-location` parameter allows an object representing the location specified in the `Path` parameter to be pushed on to the specified stack. The following command pushes a location on to the stack named `Fred` and changes the current location to `C:\PowerShellScripts`. Since the `PassThru` parameter is specified, an object is created that can be passed along the pipeline.

```
push-location -Path C:\PowerShellScripts -StackName Fred -PassThru |
get-member
```

As you can see in Figure 15-7, the object passed along the pipeline is a `PathInfo` object. You can manipulate it in any way you want. In this example, I simply display its properties using the `format-list` cmdlet. Notice, too, that the `PathInfo` object refers to the new location you have moved to, not the one you moved from.

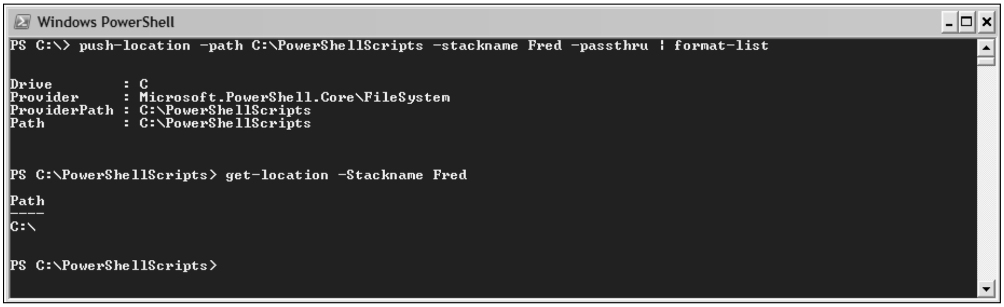


Figure 15-7

Notice, too, in Figure 15-7 that the old location has been pushed on to stack named `Fred`.

If you empty a stack and then attempt to execute the pop-location cmdlet, you will see an error message similar to the following one:

```
Pop-Location : Cannot find location stack 'Fred'. It does not exist or it is not a
container.
At line:1 char:13
+ pop-location  <<<< -stackname Fred
```

The pop-location Cmdlet

The `pop-location` cmdlet allows you to pop a location from the default location stack or a named location stack. The location popped from the stack, whether it's the default stack or a named stack, becomes the current working directory. In addition to supporting the common parameters, the `pop-location` cmdlet supports the following parameters:

- ❑ `StackName` — Specifies the name of the stack to be used
- ❑ `PassThru` — Passes an object corresponding to the current working location along a pipeline

Both parameters are optional.

The following command pops the location `C:\` from the stack `Fred`. First, you may want to confirm the locations on the stack `Fred` using:

```
get-location -StackName Fred
```

Then pop the location `C:\` from the stack using the command:

```
pop-location -StackName Fred -PassThru |
format-list *
```

The presence of the `PassThru` parameter allows you to work with the `PathInfo` object representing the location popped from the stack downstream in the pipeline. Figure 15-8 shows the results.

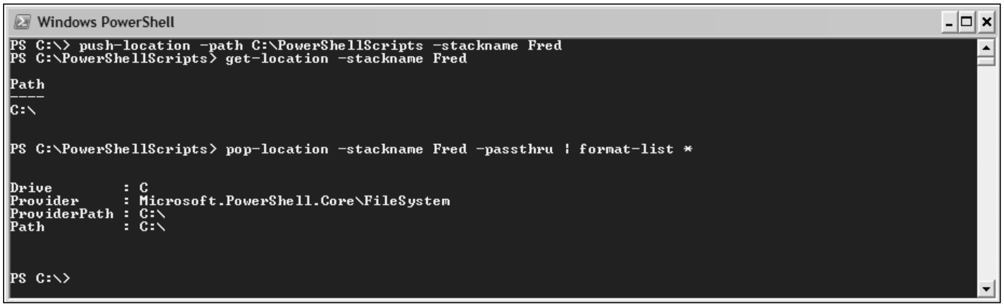


Figure 15-8

Handling Errors

Windows PowerShell has several built-in variables which define what happens when an error occurs. You can view basic information about the error-related variables by typing:

```
get-childitem variable:*error* |
format-table Name, Description -auto
```

Figure 15-9 shows the results.

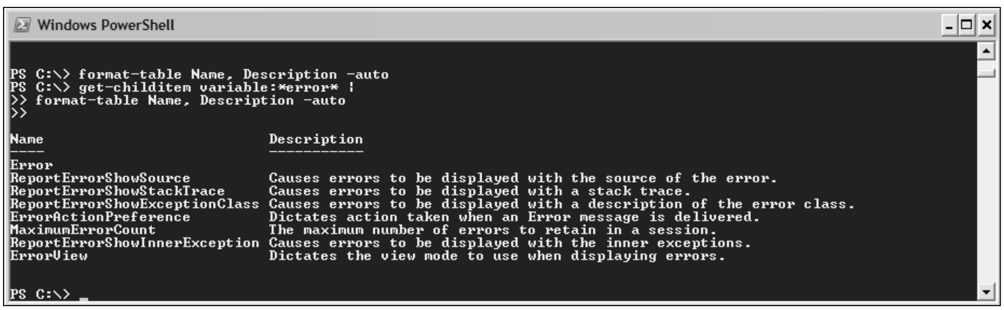


Figure 15-9

If you want to see exhaustive information about the error-related variables use the following command:

```
get-childitem variable:*error* |
format-list
```

I discuss errors and how you handle them in Chapter 17, so I won't discuss this topic in detail here.

Namespaces

Windows PowerShell namespaces provide a way to ensure that names are unique. The namespaces correspond to the standard PowerShell providers listed in the following table.

Standard Provider	Description
`Alias`	Provides access to the defined aliases
`Certificate`	Provides access to defined certificates
`Environment`	Provides access to Windows environment variables
`FileSystem`	Provides access to Windows drives and files
`Function`	Provides access to all defined functions
`Registry`	Provides access to the `HKLM` and `HKCU` hives of the registry
`Variable`	Provides access to all defined variables

I discuss the `Certificate` namespace in Chapter 16. I discuss working with the registry in Chapter 20. I discuss working with environment variables in Chapter 21.

PowerShell Aliases

Another aspect of how your system behaves is the set of aliases that have been defined for Windows PowerShell. The `Alias` drive lists all available aliases. To display the contents of the `Alias` drive, type this:

```
get-childitem alias:*
```

To navigate to the Alias drive, type:

```
cd Alias:
```

You must include the colon character after the name of the drive to successfully navigate to the `Alias` drive or an error message will be displayed.

If the `Alias` drive is the current drive, you need only type the following command to display all child items of the `Alias` drive:

```
get-childitem
```

To find the available aliases for a specific cmdlet, use the `where-object` cmdlet to filter the child items of the `Alias` drive. For example, to find the aliases for the `get-process` cmdlet, use this command (assuming that the Alias drive is the current drive):

```
get-childitem |
where-object {$_.Definition -eq "get-process"}
```

or:

```
get-childitem alias: |
where-object {$_.Definition -eq "get-process"}
```

which will work whatever the current drive.

To find all aliases for a related group of cmdlets, use the `match` operator with the `where-object` cmdlet. For example, to find all cmdlets that have `process` as their noun part, use this command:

```
get-childitem |
where-object {$_.Definition -match ".*-process"}
```

The expression in the second step of the pipeline uses a regular expression pattern, `.*-process`. The period, the first character in this regular expression, indicates a pattern which matches zero or more characters. The hyphen matches literally, as does the character sequence `process`. In other words, the pattern means find a match where there are zero or more characters followed by a hyphen followed by `process`. In the context of the value of a `Definition` property, the pattern matches any alias whose cmdlet has `process` as its noun part. Figure 15-10 shows the results.

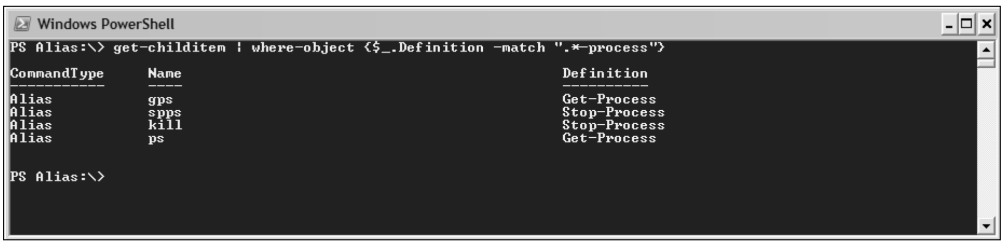


Figure 15-10

Similarly, to find all aliases that have `stop` as the verb part of the corresponding cmdlet, use this command:

```
get-childitem |
where-object {$_.Definition -match "stop-*"}
```

Again, the interesting part of the command is the regular expression, which is the operand to the `match` operator. The pattern `stop-*` means that any value for the `Definition` property which begins with the literal character sequence `stop-` will be matched. Figure 15-11 shows the results.

Figure 15-11

There are several cmdlets that allow you to work with aliases:

- `export-alias` — Exports a list of aliases to a file
- `get-alias` — Finds the cmdlet corresponding to an alias
- `import-alias` — Imports a list of aliases from a file
- `new-alias` — Creates a new alias-cmdlet pairing
- `set-alias` — Creates a new alias-cmdlet pairing or changes the association between an existing alias and its cmdlet or other element

When you create an alias you might want to specify options about its scope and whether or not it can be changed. The `-option` parameter that is available on both the `new-alias` and `set-alias` cmdlets allows you to set options. The supported values are:

- `None` — sets no options. The default.
- `Readonly` — You can change the alias only using the `-force` parameter of the `set-alias` cmdlet. You can delete the alias by using the `remove-item` cmdlet.
- `Constant` — You cannot delete the alias nor can its properties be changed. You can specify the `Constant` option only when you create an alias. You can't use the `set-alias` cmdlet to set the option to `Constant` for an existing alias.
- `Private` — The alias is available only in a scope specified by the value of the `-scope` parameter.
- `AllScope` — The alias is copied to any new scopes that are created and so is available in all scopes.

The following command creates an alias that is `Constant;` in other words that you won't be able to delete in the PowerShell session:

```
new-alias CantDeleteMe clear-host -option Constant
```

You can't delete that alias until you shut down that PowerShell session.

If you have aliases that you want to make available for all PowerShell sessions (either for one user or all users), then you can add the `new-alias` or `set-alias` commands with the `Constant` option to the relevant profile file.

Be careful that you don't make any spelling mistakes when using the `Constant` option, as in the following code (`clear-hsot` instead of `clear-host`), particularly when you use a command in a profile file. You can have a situation where the alias has been created (seemingly successfully since no error message is displayed), but can't be deleted and also doesn't work because you have misspelled the cmdlet name.

```
new-alias CantDeleteMe2 clear-hsot -option Constant
```

If you attempt to use the `CantDeleteMe2` alias, you will see the following error message:

```
Cannot resolve alias 'CantDeleteMe2' because it refers to term 'clear-hsot', which
is not recognized
t, function, operable program, or script file. Verify the term and try again.
At line:1 char:13
+ CantDeleteMe2 <<<<
```

If you attempt to delete it, you see this error message:

```
Remove-Item : Alias was not removed because alias CantDeleteMe2 is constant and
cannot be removed.
At line:1 char:12
+ remove-item  <<<< alias:CantDeleteMe2
```

Your only option (assuming that you need the alias, for example if scripts depend on it being there) is to close the PowerShell console and relaunch it. If the error is in a profile file, you also need to make the necessary edit(s) there, too.

PowerShell Functions and Filters

Functions and filters are contained in the `Function` drive. A function is a named block of code that you can execute by referring to its name. A filter is a named block of code intended, for example, as the content of the script block of the `where-object` cmdlet. To find the available functions and filters, type this command:

```
get-childitem function:*
```

The available functions and filters depend on which function declarations are included in any profile files that are executed when Windows PowerShell is starting up on your machine.

To view the definition of a function, you make use of its `Definition` property. To specify the function of interest use the `get-childitem` cmdlet with the argument `function:functionName`. For example to display the definition of the `Prompt()` function, use either of these commands:

```
get-childitem function:prompt |
format-list
```

or:

```
(get-childitem function:prompt).definition
```

Figure 15-12 shows how the function definition is displayed using each of the preceding commands.

Figure 15-12

PowerShell Variables

Variables are contained in the `Variable` drive. To view all available variables, type this command:

```
get-childitem variable:*
```

The `get-childitem` cmdlet finds the child items of a specified location. The location is specified as the variable drive. The * wildcard indicates that all child items (that is all variables) are of interest.

To display information about a specified variable, supply its name after the drive name. For example, to display information about the `$Profile` variable, type this command. Be careful not to include the `$` sign when specifying the name of interest.

```
get-childitem variable:Profile
```

Or, if you prefer fuller information about the variable, use this command:

```
get-childitem variable:Profile |
format-list
```

Figure 15-13 shows the results of running the two commands.

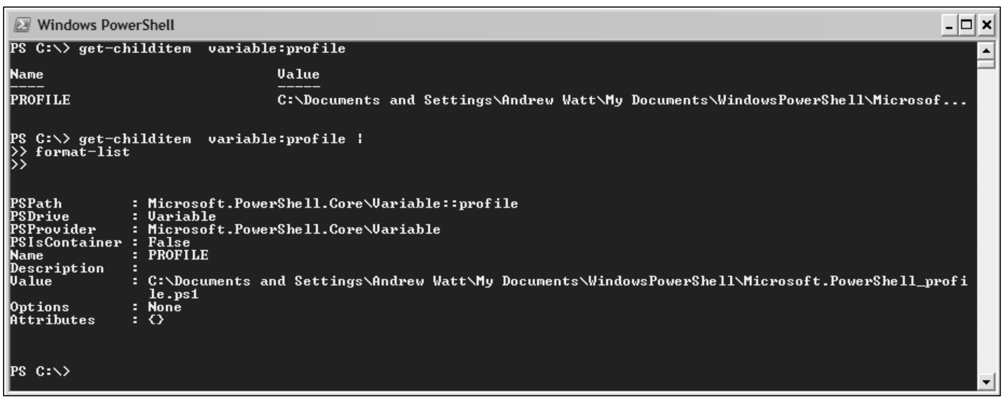


Figure 15-13

Exploring the Environment Variables

The `Environment` provider provides read/write access to Windows system environment variables from within Windows PowerShell. The provider exposes a single env: drive. The environment variables are exposed as if they belonged to any conventional drive. So, just as you can use the `get-childitem` cmdlet or its alias `dir` to explore conventional drives that use the `FileSystem` provider, you can also use the `get-childitem` cmdlet to retrieve information about environment variables. To find out what the currently set environment variables are and sort them alphabetically, type the following command:

```
get-childitem env:*
sort-object Name
```

If env: is already the selected drive then simply type:

```
get-childitem *
sort-object Name
```

To find a named environment variable, for example the `UserName` environment variable, use the `Path` parameter with the `get-childitem` cmdlet:

```
get-childitem env:UserName
```

Figure 15-14 shows the result of executing the preceding command.

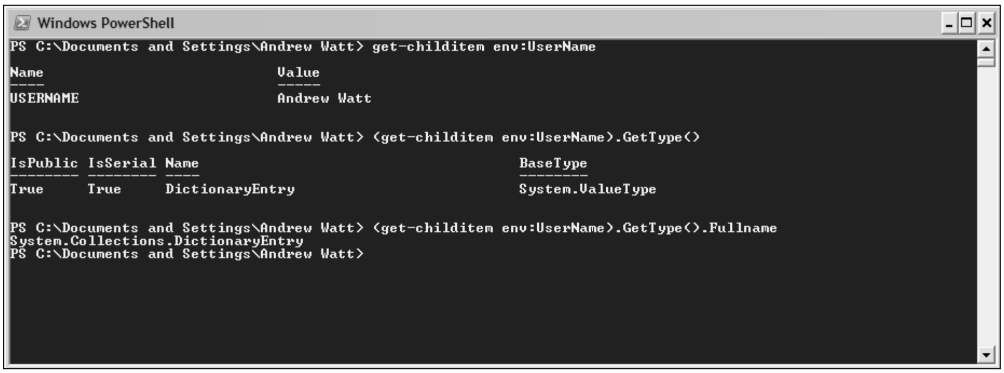


Figure 15-14

In Windows PowerShell, each environment variable is a `System.Collections.DictionaryEntry` object, as you can see in the lower part of Figure 15-14 or by running either of the following commands:

```
(get-childitem env:UserName).GetType()
```

or:

```
(get-childitem env:UserName).GetType().Fullname
```

Using the `get-childitem` cmdlet to display an environment variable works well if the value of the `Value` property is short. For some environment variables, the preceding approach fails to display all of the `Value` property. For example, the value of the `Path` environment variable is often long. Use the `format-list` cmdlet to display the full value of the `Value` property, as in the following command:

```
get-childitem env:Path |
format-list
```

Figure 15-15 shows the results displayed by the two approaches.

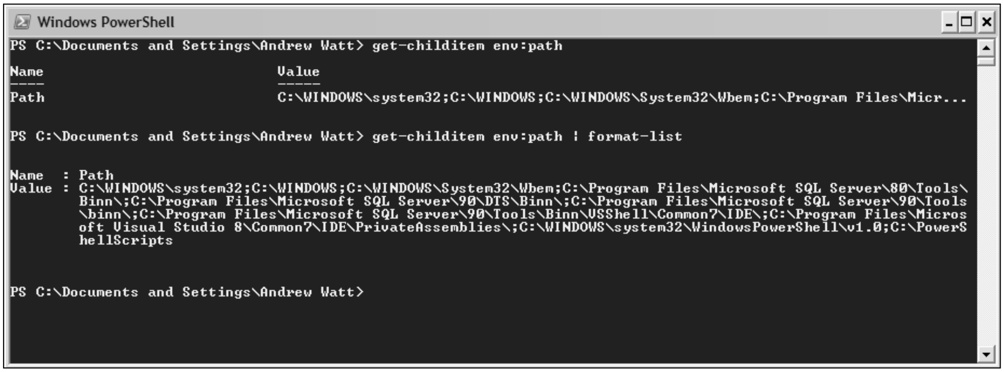


Figure 15-15

Environment variables in PowerShell are variables. So, you can assign values to them just as you would other PowerShell variables.

To change the value of the `$env:UserProfile`, simply assign another value to it. The following command modifies the `$env:UserProfile` variable:

```
$env:UserProfile = "C:\Documents and Settings"
```

Figure 15-16 shows the variable being changed and changed back to its original value.

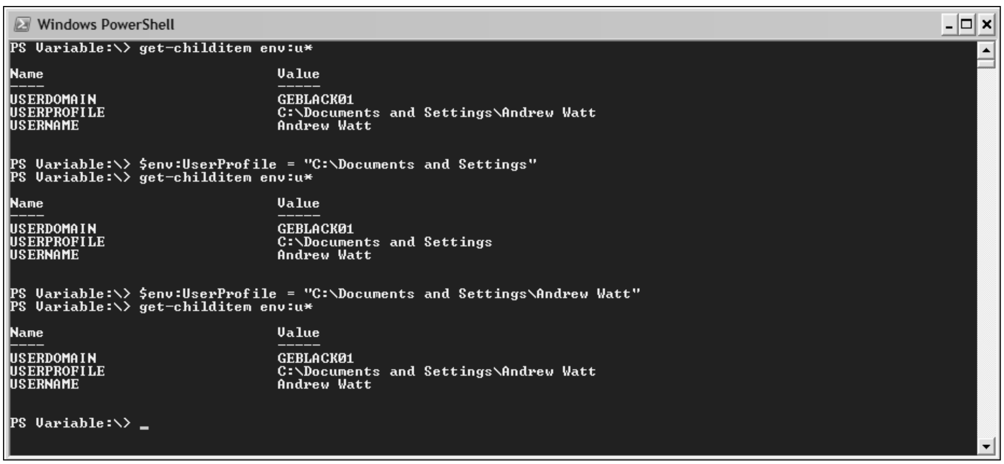


Figure 15-16

Before experimenting with the Path variable, you might want to use the following command, so that you can later restore the value of the $env:Path variable.

```
$env:path = $oldPath
```

The following command displays each folder in the $env:Path variable:

```
($env:Path).Split(';')
```

You can use the += assignment operator to add a folder to the value of $env:Path:

```
$env:path += ";C:\Disposable"
```

Notice that a semicolon is the first character in the value assigned, since examining the existing value of the $env:Path variable showed there was no terminating semicolon in the value.

Figure 15-17 shows an example of changing the value of the $env:path variable using the += assignment operator.

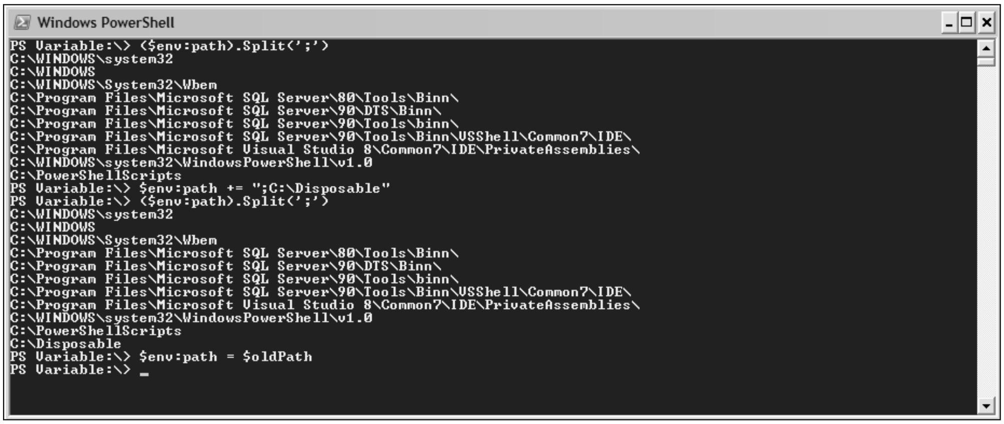


Figure 15-17

At the end, the original value of the $env:Path variable was restored using the command:

```
$env:Path = $oldPath
```

Exploring the Current Application Domain

Another aspect of the environment that Windows PowerShell allows you to explore is the application domain. An application domain is represented by the .NET System.AppDomain class. An application domain provides isolation, unloading, and security boundaries for executing managed code. This can be useful where a software module is running in relation to a process. If the software module crashes but is running in a separate application domain, it is possible to unload the application domain with the crashed software module without adversely affecting the execution of the process.

You can find out the current application domain using the `get_CurrentDomain()` method or the `CurrentDomain` property of the `System.AppDomain` class. Either of the following commands assigns an object representing the current domain to the variable `$CurrDomain`:

```
$CurrDomain = [System.AppDomain]::get_CurrentDomain()
```

or:

```
$CurrDomain = [System.AppDomain]::CurrentDomain
```

To display basic information about the current application domain, you can simply type:

```
$CurrDomain
```

Figure 15-18 shows the information about the current application domain on one of the computers I am using to write this book.

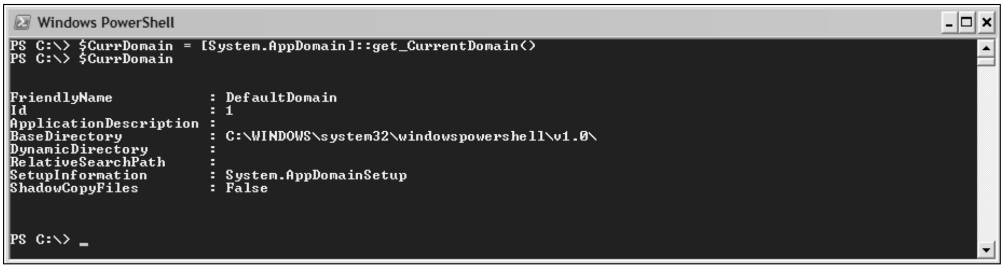


Figure 15-18

As always, you can find the members of an object using the `get-member` cmdlet. The following command finds the members of the `$CurrDomain` object:

```
$CurrDomain |
get-member
```

Among the members of the `$CurrDomain` object is the `GetAssemblies()` method, which allows you to find the assemblies that have been loaded in the current application domain. A convenient way to display a list of the assemblies used by the current application domain is this command:

```
$CurrDomain.GetAssemblies() |
format-table Fullname
```

Figure 15-19 shows the assemblies of the default application domain on a Windows XP machine.

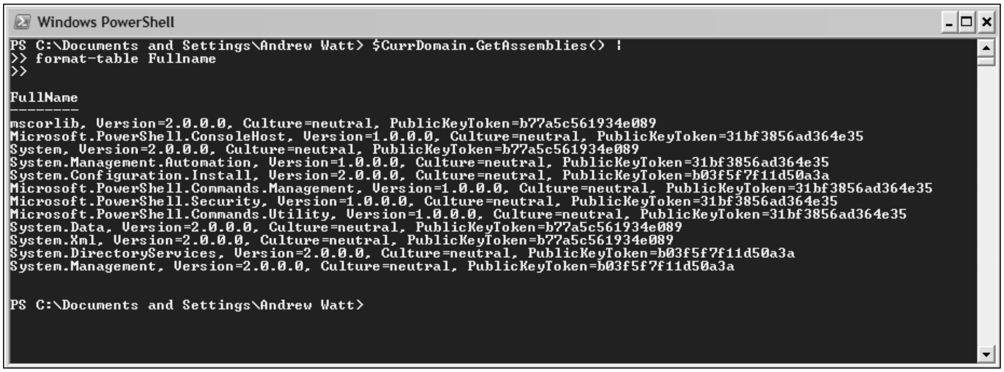


Figure 15-19

As you can see in Figure 15-19, one of the assemblies loaded in the current application domain is `mscorlib`. You might want to explore the core library to, for example, find out how many types it defines and what those types are. To do that, use this command:

```
$CoreLib = $CurrDomain.GetAssemblies() |
where-object {$_.Fullname -match "mscorlib"}
```

to assign the mscorlib assembly to the `$CoreLib` variable. The second step of the pipeline filters the assemblies using the `Fullname` property of each current object in turn and determines whether it contains the sequence of characters `mscorlib`, which, as you saw in Figure 15-19, is part of the value of the `Fullname` property of the core library.

To display basic information about the core library, use this command:

```
$CoreLib | format-list
```

As you can see in Figure 15-20, the `mscorlib.dll` file contains the core library. You can also see the location of the DLL.

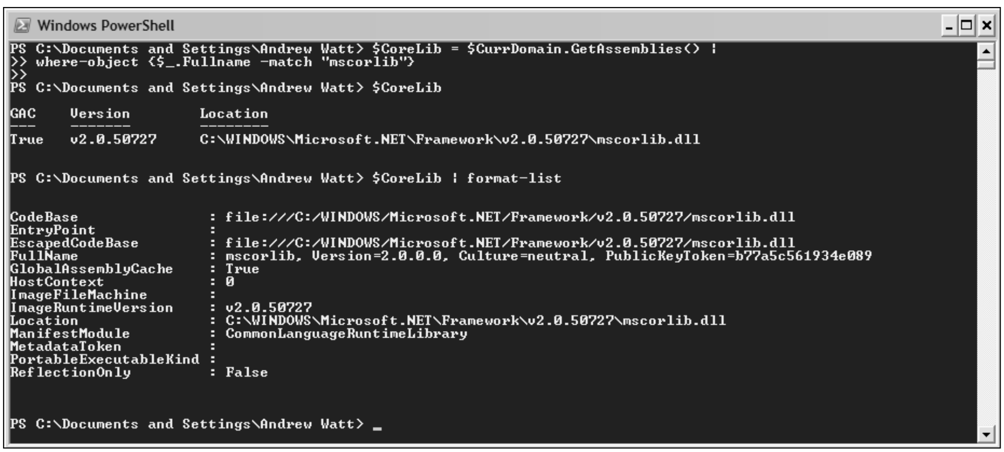


Figure 15-20

To find the number of types defined by the core library, use this command:

```
$($CoreLib.GetTypes()).Count
```

The contents of the paired parentheses, `$CoreLib.GetTypes()`, is evaluated first, then the `Count` property of that result is displayed. On the machine that I was using, 2,319 types were defined in the core library. When you have very large numbers of results, as with the preceding command, you may wish to filter the results using the `where-object` cmdlet, or use the `group-object` cmdlet to see which groups the results belong to.

To display the namespaces those types belong to, use this command:

```
$CoreLib.GetTypes() |
group-object Namespace |
format-table -auto
```

As you can see in Figure 15-21, there are many namespaces used.

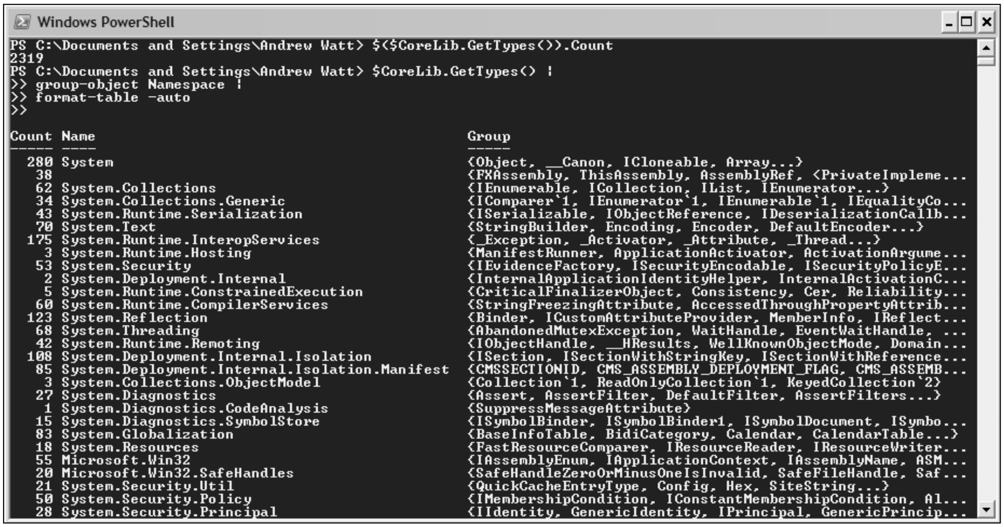


Figure 15-21

If you want to better understand what is happening in .NET 2.0 under the covers, you might want to explore which classes in the `System.Resources` namespace are defined in the core library. To do that, use this command:

```
$CoreLib.GetTypes() |
where-object {$_.Namespace -eq "System.Resources"}
```

The version number of the core library can be used to check whether all machines have a desired version of the .NET Framework available. At the moment, the only version of the .NET Framework 2.0 supported by Powershell V1 is .NET 2, or version 2.0.0.0. To check whether a machine has that version loaded, use this command:

```
$CoreLib.Fullname -match "2.0.0.0"
```

The `match` parameter indicates that a regular expression is to be matched. Since there are no *metacharacters* matching the start or end of a string, the literal pattern `2.0.0.0` effectively matches any string that includes that sequence of characters.

A metacharacter is part of a regular expression that has a meaning other than its literal meaning. For example, ^ matches the position at the start of a sequence of characters but does not match any of the characters.

Be aware that version 2.0.0.0 of `mscorlib.dll` corresponds to the version of the .NET Framework 2.0 in the folder `C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727`. Just be aware that they refer, using different numbering systems, to the same version of the .NET Framework.

Later in the evolution of .NET 2.x you may want to check that some later version is loaded. You can modify the preceding command to determine that. By including that test in a script you can, for example, list all machines without the desired version of the .NET Framework installed.

The preceding examples are simply that — examples of a huge range of possibilities of how you might use Windows PowerShell to explore the environment that it is running in.

Exploring Services

Using Windows PowerShell to explore what services are installed or running on your machine is straightforward using the Windows PowerShell cmdlets designed to retrieve information about services or to modify their behavior.

The following cmdlets relevant to services are supported by Windows PowerShell version 1.0:

- `get-service` — Retrieves a list of services
- `new-service` — Creates a new service
- `restart-service` — Restarts a stopped service or stops and restarts a running service
- `resume-service` — Resumes a suspended service
- `set-service` — Makes changes to the properties of a service
- `start-service` — Starts a stopped service
- `stop-service` — Stops a running service
- `suspend-service` — Suspends a running service

Using the get-service Cmdlet

The `get-service` cmdlet retrieves information about one or more services. In addition to the common parameters, the `get-service` cmdlet supports the following parameters (all the listed parameters are optional):

- `Name` — Specifies the name(s) of the service(s) to be retrieved. Cannot be used with the `DisplayName` parameter.
- `Include` — Specifies those items on which the cmdlet will act.
- `Exclude` — Specifies those items on which the cmdlet will not act.
- `DisplayName` — Specifies the display name(s) of the service(s). Cannot be used with the `ServiceName` parameter.
- `InputObject` — The `ServiceController` object for the service(s) about which you want to retrieve information.

To retrieve information about all services installed on a system, simply type the following command without specifying any parameters:

```
get-service
```

You can focus the objects returned by specifying the `-Name` parameter with a suitable argument. You can use wildcards in the value of the `-Name` parameter. For example, to retrieve information about all services whose service name begins with `m`, use the command:

```
Get-Service -Name m*
```

To display those services sorted alphabetically and with their display name, use this command:

```
get-service -ServiceName m* |
sort-object ServiceName |
format-table Name,DisplayName -auto
```

Figure 15-22 shows the results of running the preceding command.

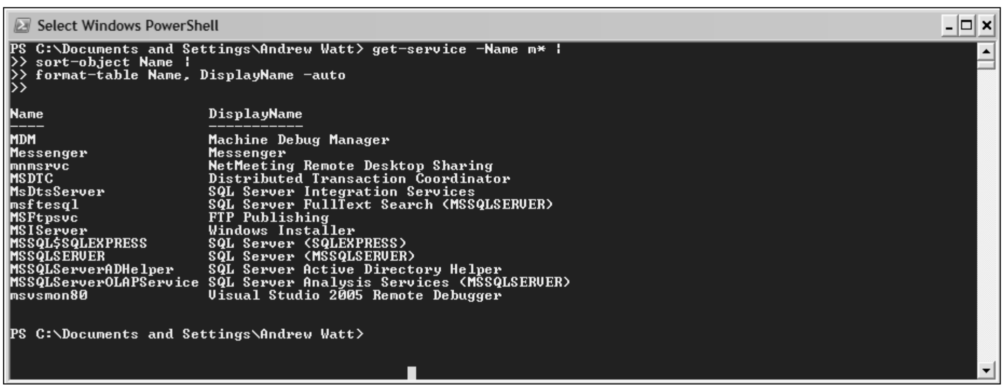


Figure 15-22

You can use the `Status` property of the objects returned from the `get-service` cmdlet together with the `where-object` cmdlet to display only running or stopped processes. For example, to retrieve information about all running services related to SQL Server 2005 (on a machine on which SQL Server 2000 or earlier is not installed), use this command:

```
get-service -Name *sql* |
where-object {$_.Status -eq "Running"} |
format-table Name, DisplayName, Status -auto
```

Figure 15-23 shows the results of executing the preceding command. If you don't have SQL Server installed, modify the value of the `-name` parameter to another value, for example, `t*`.

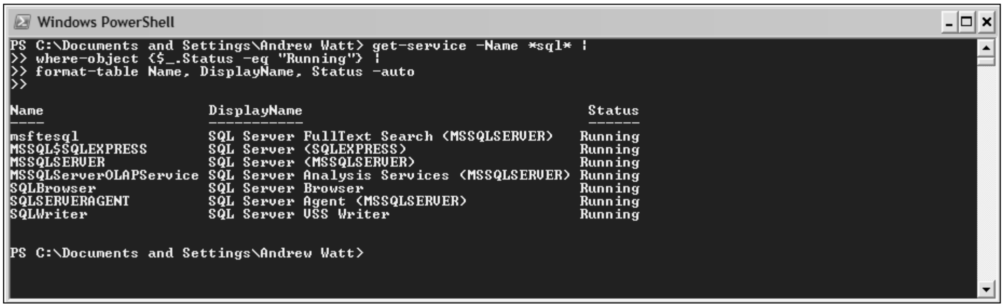


Figure 15-23

To display all stopped SQL Server 2005 services, use this command:

```
get-service -Name *sql* |
where-object {$_.Status -eq "Stopped"} |
format-table Name, DisplayName, Status -auto
```

The `-exclude` parameter filters service objects as specified by the `-Name` parameter. The value of the `-exclude` parameter can include wildcards. For example, to find all services but exclude those whose service names begin with `a` through `f` and `n` through `z`, use the following command:

```
get-service -Name * -Exclude [a-fn-z]* |
sort-object Name |
format-table Name, DisplayName -auto
```

Figure 15-24 shows the results of running the preceding command.

Figure 15-24

Using the new-service Cmdlet

The `new-service` cmdlet allows you to create a new service. The `new-service` cmdlet supports the following parameters in addition to the ubiquitous parameters:

- `Name` — The name of the new service to be created. A required, positional parameter in position 1.
- `BinaryPathName` — The path to the executable file of the service to be created. A required, positional parameter in position 2.
- `DisplayName` — The display name for the new service.
- `Description` — A description of the new service.
- `StartupType` — Specifies how the service behaves at system startup. Values are enumerated. Allowed values are `Automatic`, `Manual` and `Disabled`.
- `Credential` — Specifies the credential that the service will start under.
- `DependsOn` — Names of other services on which the new service depends.

The following command creates an entry in the registry and in the Service database for a service named `MyLanService2`:

```
new-service -Name MyLanService2 -BinaryPathName "C:\Windows\System32\svchost.exe -k
netsvcs"
```

If the new service is successfully registered, its status, service name, and display name are echoed to the console, as shown in Figure 15-25.

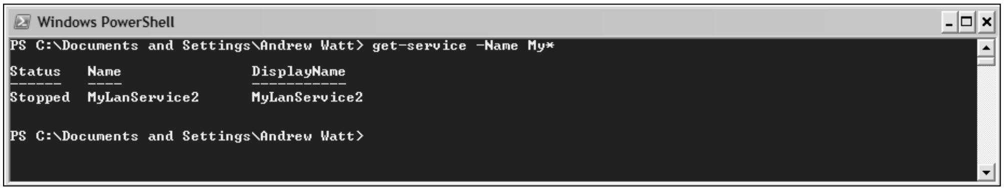


Figure 15-25

Using the restart-service Cmdlet

The `restart-service` cmdlet is used to start or restart a service. That is, if the target service is stopped, the `restart-service` cmdlet starts it. If the service is running, the `restart-service` cmdlet stops the service, then restarts it.

The `restart-service` cmdlet supports the following parameters in addition to the common parameters:

- `Name` — The name of the service to be restarted. This is a required parameter, which is a positional parameter in position 1.
- `Include` — Specifies those items on which the cmdlet will act. Wildcards are allowed.
- `Exlude` — Specifies those items on which the cmdlet will not act. Wildcards are allowed.
- `PassThru` — The object relating to the restarted service is passed along the pipeline.
- `Force` — Force a restart of services dependent on the service to be restarted.
- `DisplayName` — The display name(s) of the service(s) to be restarted.
- `InputObject` — A `ServiceController` object for the service to be restarted.
- `Whatif` — Describes what would happen if the command were executed. No changes are actually made.
- `Confirm` — Prompts for confirmation before executing the command.

The following command will stop and restart (or just start) the `w3svc` service:

```
restart-service w3svc
```

If you stop the `w3svc` using the command

```
stop-service w3svc
```

you can then use the command

```
restart-service w3svc
```

to restart the service.

Using the set-service Cmdlet

The `set-service` cmdlet allows you to change the properties of a service. In addition to the common parameters, the `set-service` cmdlet supports the following parameters:

- `Name` — The name of the service whose properties are to be modified.
- `DisplayName` — The display name of the service whose properties are to be modified.
- `Description` — A description of the service whose properties are to be modified.
- `StartupType` — Specifies how the service behaves on system startup. Allowed values are an enumeration: `Disabled`, `Manual`, or `Automatic`.
- `Whatif` — Describes what would happen if the command were executed. No changes are actually made.
- `Confirm` — Prompts for confirmation before executing the command.

The following command changes the display name of the service named `MyLanService`:

```
set-service -ServiceName MyLanService2 -DisplayName "My Lan Service"
```

Figure 15-26 shows the results of executing the preceding command. Notice that after execution of the command, spaces have been introduced into the display name.

Figure 15-26

Using the start-service Cmdlet

The `start-service` cmdlet is used to start a service. If the service is already running the command is ignored without error. It supports the following parameters in addition to supporting the common parameters:

- `Name` — The name(s) of the service(s) to be started.
- `Include` — Specifies those items on which the cmdlet will act.
- `Exclude` — Specifies those items on which the cmdlet will not act.
- `PassThru` — The object created is made available to the pipeline.
- `DisplayName` — The display name of the service to be started.

- `InputObject` — An array of `ServiceController` object for the service(s) to be started.
- `Whatif` — Describes what would happen if the command were executed. No changes are actually made.
- `Confirm` — Prompts for confirmation before executing the command.

To start the `w3svc` service, if it is stopped, use the following command:

```
start-service w3svc
```

As you can see in Figure 15-27, the `w3svc` service was stopped before the preceding command was run and was running after the command was issued.

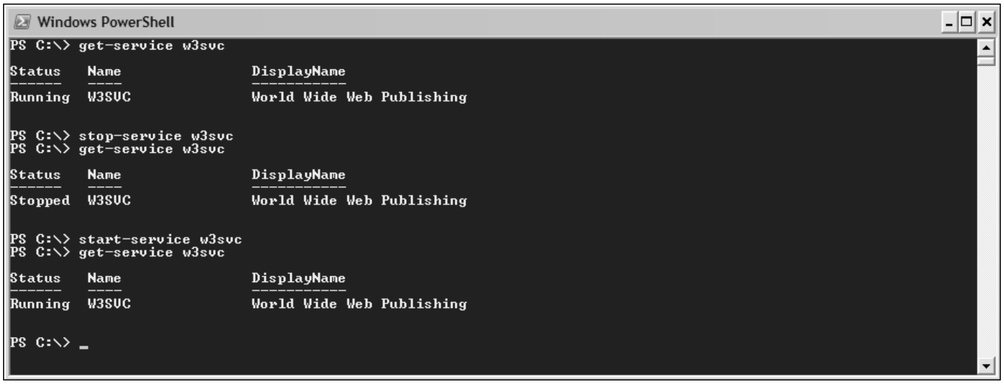


Figure 15-27

Using the stop-service Cmdlet

The `stop-service` cmdlet stops named service(s). The `stop-service` cmdlet supports the following parameters in addition to the ubiquitous parameters:

- `Name` — The name(s) of the service(s) to be stopped.
- `Include` — Specifies those items on which the cmdlet will act.
- `Exclude` — Specifies those items on which the cmdlet will not act.
- `Force` — Allows the cmdlet to override dependency restrictions.
- `PassThru` — The object created is passed to the next step in the pipeline.
- `DisplayName` — The display name(s) of the service(s) to be stopped.
- `Input-Object` — An array of `ServiceController` object for the service(s) to be stopped.
- `Whatif` — Describes what would happen if the command were executed. No changes are actually made.
- `Confirm` — Prompts for confirmation before executing the command.

To stop the `w3svc` service, use this command:

```
stop-service -Name w3svc
```

Using the suspend-service Cmdlet

The `suspend-service` cmdlet suspends a running service. Not all services support being suspended. In addition to supporting the ubiquitous parameters, the `suspend-service` cmdlet supports the following parameters:

- `ServiceName` — The name(s) of the service(s) to be suspended.
- `Include` — Specifies those items on which the cmdlet will act.
- `Exclude` — Specifies those items on which the cmdlet will not act.
- `DisplayName` — The display name(s) of the service(s) to be suspended.
- `Force` — Allows the cmdlet to override dependency restrictions.
- `PassThru` — The object created is passed to the next step in the pipeline.
- `InputObject` — An array of `ServiceController` object for the service(s) to be suspended.
- Whatif — Describes what would happen if the command were executed. No changes are actually made.
- Confirm — Prompts for confirmation before executing the command.

To suspend the `w3svc` service, use this command:

```
suspend-service -ServiceName w3svc
```

Figure 15-28 shows the preceding command pausing the `w3svc` service.

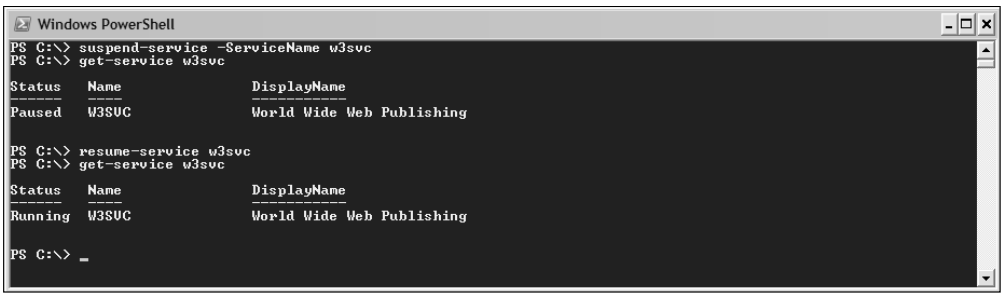


Figure 15-28

To restart a suspended service you use this command:

```
restart-service -ServiceName w3svc
```

Attempting to start a suspended service using `start-service` does not work.

Summary

Windows PowerShell allows you to explore aspects of the system on which it is running. You can work with locations in multiple providers. I introduced the following cmdlets:

- `get-location` — Returns the current location
- `push-location` — Pushes a location on to the default stack or a named stack
- `pop-location` — Retrieves a location from the default stack or a named stack

You can explore aliases, functions, variables, and environment variables.

PowerShell provides several cmdlets that allow you to work with services that are registered on a system:

- `get-service` — Allows you to retrieve information about registered services
- `new-service` — Registers a service
- `restart-service` — Restarts a service
- `set-service` — Changes one or more properties of a service
- `start-service` — Starts a service
- `stop-service` — Stops a service
- `suspend-service` — Pauses a service
- `restart-service` — Restarts a paused service

16

Security

If you have worked through earlier chapters of this book, you will have begun to understand the huge potential that Windows PowerShell has for inspecting and manipulating Windows computers. Any software that allows you to discover what is happening on a system and modify that system and what is stored in its files has enormous power. That gives you power to do good. But with power also comes risk.

The designers of Windows PowerShell have spent significant time to analyze those risks. As a result, Windows PowerShell has an execution policy that, by default, prevents you running any PowerShell scripts. This is part of an approach that Microsoft calls Secure by Default. When you install the product, it is intended to be secure. This means that you need to take active steps to enable features that you want. In PowerShell executing scripts is a prominent example.

What is the reason for the Secure by Default approach? Imagine the scenario where you have just installed PowerShell and downloaded a script from the Internet or are sent a script by an acquaintance. With your possibly limited understanding of PowerShell, the risk of your running a malicious script has to be there. That script could remove files from your hard drive or run other scripts, and those scripts, in turn, could be malicious. The potential for damage is obvious. The security policies for Windows PowerShell are designed to allow you to configure security intelligently once you understand the implications of your actions, so that you find the appropriate balance between security and functionality for your business scenario(s).

If you're going to be able to persuade your managers that installing Windows PowerShell widely is a safe thing to do, then you need to understand what protections are in place and how to make the appropriate adjustments to address your company's business scenario.

Minimizing the Default Risk

When you install Windows PowerShell, there are several factors that reduce the chances of any malicious script being run. By default, immediately after you install PowerShell you can't run Windows PowerShell scripts at all.

If, with a default installation of Windows PowerShell, you attempt to run a Windows PowerShell script (in any of a number of ways, as described in this section), you run into a brick wall. Double-clicking on a script in Windows Explorer causes the script to open in Notepad

Similarly, if you attempt to run a script by right-clicking on it and choosing the Open option from the context menu, the script is again opened in Notepad.

Right-clicking on a script and selecting the Open With option, then selecting Windows PowerShell from C:\WINDOWS\system32\windowspowershell\v1.0 fails to add Windows PowerShell to the list of programs that can open a `.ps1` file. I also tried copying the shortcut to Windows PowerShell to the desktop and then tried to add that shortcut to the list of programs to open the script, but the error message shown in Figure 16-1 was displayed.

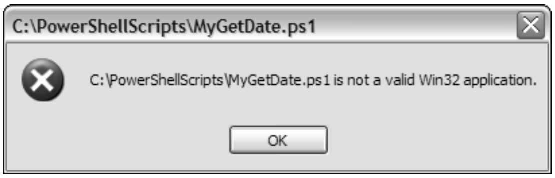


Figure 16-1

I couldn't find a way to make a PowerShell script execute from Windows Explorer. That is a good thing, since it protects each Windows machine from an innocent user double-clicking on a file and executing a malicious script. This protection is permanent. As far as I'm aware there is no way of working round it.

The default behavior of opening a text editor seems to depend on a key value located at `HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.ps1`. The value of `PerceivedType` is `Text`. If you modify that then a `.ps1` script no longer opens in Notepad by default when it is double-clicked.

If you can't open a Windows PowerShell script from Windows Explorer, what's the default situation on the PowerShell command line? To demonstrate that I created a very simple one line script:

```
write-host "Hello world! This script has run."
```

and stored it as `Hello.ps1`.

If you open a Windows PowerShell command shell, navigate to the script's directory and then attempt to run it from the command line, you will again hit barriers. If you simply type the filename:

```
Hello.ps1
```

on the command line, an error message that the script file `Hello.ps1` is not recognized as a script file is displayed, as shown in the upper part of Figure 16-2.

```
The term 'Hello.ps1' is not recognized as a cmdlet, function, operable program, or
script file. Verify the term
 and try again.
At line:1 char:9
+ Hello.ps1 <<<<
```

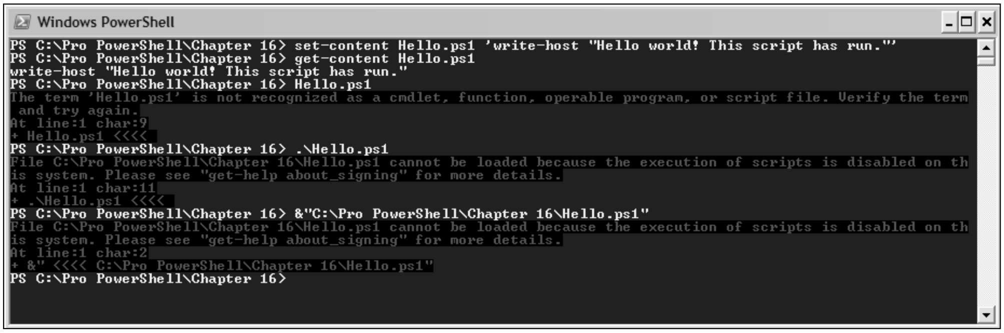


Figure 16-2

If you're aware that you can never run Windows PowerShell script by simply typing its filename, you might well try the correct syntax:

```
.\Hello.ps1
```

but that too results in an error message being displayed:

```
File C:\Pro PowerShell\Chapter 16\Hello.ps1 cannot be loaded because the execution
of scripts is disabled on th
is system. Please see "get-help about_signing" for more details.
At line:1 char:11
+ .\Hello.ps1 <<<<
```

as shown in the lower part of Figure 16-2. The latter error message tells you that execution of scripts is disabled on the machine (which it is) and gives you the first hint of what needs to be altered if you want to run Windows PowerShell scripts.

Attempting to run the script using its full path and name:

```
&"C:\Pro PowerShell\Chapter 16\Hello.ps1"
```

also won't run the script and gives the same error message.

The bottom line is that a default install of Windows PowerShell won't run *any* Windows PowerShell scripts. Not only is it not possible to execute your own scripts from the command line, but you can't execute profile files when you start a Windows PowerShell session. On one test machine, I have a personal profile that starts a transcript of my session. If I attempt, with the default settings, to open a Windows

PowerShell session, the console opens but the profile file isn't executed. The following error message is displayed:

```
File C:\Documents and Settings\Andrew Watt\My
Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1 cannot be loaded
because the execution of scripts is disabled on this system. Please see "get-help
about_signing" for more details.
At line:1 char:2
+ .  <<<< 'C:\Documents and Settings\Andrew Watt\My
Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1'
```

That's great from a security point of view. A user can't execute scripts and can't inadvertently execute a malicious profile file. But it's pretty inconvenient if you want to make use of the functionality that Windows PowerShell scripts offer you.

At the time of writing, it looks as if the security settings for a default install of Windows PowerShell are stable. However, if you have problems running all scripts that the following paragraphs don't solve for you, type `get-help about_signing`*, which opens the file* `about_signing.help.txt` *in Notepad. Check for any last minute changes in the default install. That file tells you about the execution policies current for the version of Windows PowerShell that you have installed.*

To help you manage the security settings for Powershell, you can set an *execution policy* to define what scripts Powershell is allowed to execute. You can set these policies either by directly editing the registry or by using the `set-executionPolicy cmdlet`.

The following table summarizes the characteristics of the four available execution policies. The execution policy when you first install Windows PowerShell 1.0 is `Restricted`. Following the table I describe the `get-executionpolicy` and `set-executionpolicy` cmdlets.

Execution Policy	Description
`Restricted`	No scripts or profile files are run, which is the default execution policy. Windows PowerShell can be run interactively from the command line.
`AllSigned`	Runs only scripts that are signed by a publisher that you trust. Protection depends on how trustworthy those you choose to trust are.
`RemoteSigned`	Runs all scripts except those that originate from applications like Microsoft Outlook, Internet Explorer, Outlook Express, and Windows Messenger. The latter's script and configuration files must be signed by someone you trust.
`Unrestricted`	Runs all scripts. You receive a warning when attempting to run a script downloaded from applications like Microsoft Outlook, Internet Explorer, Outlook Express, and Windows Messenger.

If you have attempted to run a script using a command like

```
.\Hello.ps1
```

and received the error message shown in Figure 16-2, you can conclude that the execution policy on the machine is `Restricted`. You can use Windows PowerShell's `get-executionpolicy` cmdlet to confirm the value of the execution policy, using the following command:

```
get-executionPolicy
```

If the execution policy is restricted, you will see a display like that in Figure 16-3. Notice that the value of the `ExecutionPolicy` property is `Restricted`.

Figure 16-3

The `get-executionpolicy` cmdlet has no parameters except the common parameters that I described in Chapter 6.

Normally, you would use the `set-executionpolicy` cmdlet to set the execution policy. It's also possible to make the necessary change directly in the registry but the `set-executionpolicy` is much more convenient to use.

`set-executionpolicy` supports the following cmdlets in addition to the common parameters (described in Chapter 6):

- `executionPolicy` — An enumeration (Restricted, AllSigned, RemoteSigned, Unrestricted, Default) that specifies the execution policy to be applied. A positional parameter in position 1.
- `whatif` — Allows you to see the change that would be made, but nothing is changed.
- `confirm` — Requires you to confirm that you want the change made.

If you use the `-whatif` parameter, you will see a message like the following, but the execution policy has not been changed:

```
What if: Performing operation "Set-ExecutionPolicy" on Target "RemoteSigned".
```

If you use the `-confirm` parameter, you will see a message like the following. Whether or not the execution policy is changed depends on your response to the message.

```
Confirm
Are you sure you want to perform this action?
Performing operation "Set-ExecutionPolicy" on Target "RemoteSigned".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is
"Y"): n
```

The execution policy that is most suited to you and your colleagues is likely to vary depending on what you want to do with Windows PowerShell, your understanding of the effects of Windows PowerShell scripts generally, your understanding of how trustworthy (or not) the signatory of a signed script is, and so on. Choose the execution policy that lets you get the job done, while still keeping your machine safe from malicious scripts.

If you are a Windows PowerShell developer, you may very well want to set the execution policy on a development machine to `Unrestricted` or `RemoteSigned`. Both settings give you some level of protection against remote malicious scripts. The `Unrestricted` setting gives you a warning message before running a remote script from the Internet but allows you the convenience to run scripts you or colleagues have written locally. If you map a drive to a local directory, the script may be treated as remote. During development and debugging, the ability to run scripts multiple times is a very definite plus, provided that the author of the script(s) is known and trusted. It would be tedious to keep having to sign scripts that are under ongoing development. The `RemoteSigned` setting allows you to run only scripts signed by those you trust. Both settings, in their different ways, tell you that there is an increased risk of running scripts written by people other than yourself (or other users of your local machine).

Be very careful when copying script fragments from the Internet (perhaps from a blog or article about a Windows PowerShell technique). If you paste such code fragments into, say, Notepad and save a script from there, it will be treated as a locally created script. If there is any malicious code in what you paste into Notepad, you are essentially unprotected from it, except for reliance on your own understanding of what the script code actually does. It is, of course, particularly dangerous to execute a complete script whose effects you don't fully understand that you copied and pasted into Notepad.

The following table makes suggestions for the execution policies for different types of users.

User Category	Suggested Setting(s)	Comment
Developer	`Unrestricted` or `RemoteSigned`	Security depends on developers understanding thoroughly what they are doing when accepting scripts or code fragments from anywhere not their own, based on their understanding of Windows PowerShell. A beginning developer needs to be sure that he understands the risk of using Windows PowerShell code, which uses the more cryptic syntax variants.
Administrator	`AllSigned` or `RemoteSigned`	Security depends on how valid trust of the signatories is. An `AllSigned` policy may be more advisable once a possible phase of local script development has stabilized.
End User	`Restricted` or `AllSigned`	Since the user may have little or no understanding of Windows PowerShell, a `Restricted` policy may be preferred.

I would emphasize that security is only as strong as the weakest link. A developer who doesn't understand that he has gaps in his understanding of Windows PowerShell is a security risk. An administrator who inappropriately alters execution policies may expose business-critical machines to malicious Windows PowerShell scripts. Enterprise-wide deployment of Windows PowerShell needs careful thought to balance the undoubted benefits of Windows PowerShell against its potential security risks.

Ultimately, security depends on multiple local factors. Not the least of which is the human factor. A developer who has set a `RemoteSigned` execution policy and has encountered no malicious scripts might be tempted by his previous experience to respond to a prompt and simply run a remotely signed script that contains malicious code.

As always when considering security, think about physical access to a local machine. For example, if an outsider has access to the machine, perhaps for repair or upgrading, it is important to check that security settings for Windows PowerShell (and other software) have not been inappropriately changed.

The bottom line is that the preceding suggestions can be only that — suggestions. You need to thoroughly understand your business scenario before making a decision about which execution policy to use for which users. Once you have given careful consideration to your security scenario you can make an intelligent choice about how and whether to modify the default execution policy.

Using the `set-executionpolicy` *cmdlet requires Administrator privileges.*

To change the execution policy to `Unrestricted`, use this command:

```
set-executionpolicy -executionPolicy Unrestricted
```

If you want to set the execution policy to `AllSigned` or `RemoteSigned`, simply vary the value of the `-executionPolicy` parameter appropriately. If you later want to reset the execution policy to `Restricted`, simply supply that as the value of the `-executionPolicy` parameter.

Figure 16-4 shows the execution policy set to `Unrestricted`. As mentioned earlier, that is a setting that should only be used by those who fully understand its implications.

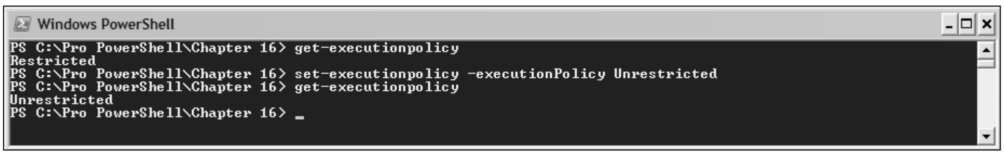


Figure 16-4

If I changed the execution policy to `AllSigned`, the unsigned script `Hello.ps1` would not run.

```
File C:\Pro PowerShell\Chapter 16\Hello.ps1 cannot be loaded. The file C:\Pro
PowerShell\Chapter 16\Hello.ps1 i
s not digitally signed. The script will not execute on the system. Please see "get-
help about_signing" for more
 details..
At line:1 char:11
+ .\Hello.ps1 <<<<
```

However, the signed script `HelloSigned.ps1` would run, but only after I was asked if I wanted to run the script. Since it was signed by a certificate signed on the local machine, it was trusted. I discuss signing scripts later in this chapter.

Figure 16-5 shows the results of attempting to run the scripts `Hello.ps1` and `HelloSigned.ps1` while the execution policy was set to `AllSigned`.

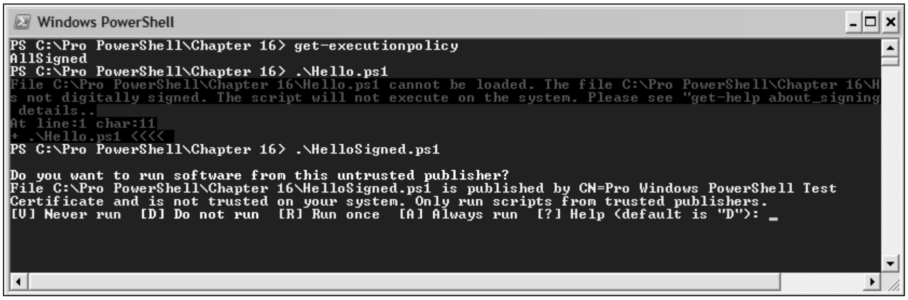


Figure 16-5

If you select the option `[R] Run once` you run the script once. If you select the option `[A]Always run`, then the script `HelloSigned.ps1` will run at future times on that machine without further user prompts about whether to trust that script, assuming that the execution policy remains unchanged.

If you then change the execution policy to `RemoteSigned`, both `Hello.ps1` and `HelloSigned.ps1` run without prompts, as shown in Figure 16-6, since neither script was downloaded from the Internet, but rather each was created on the local machine.

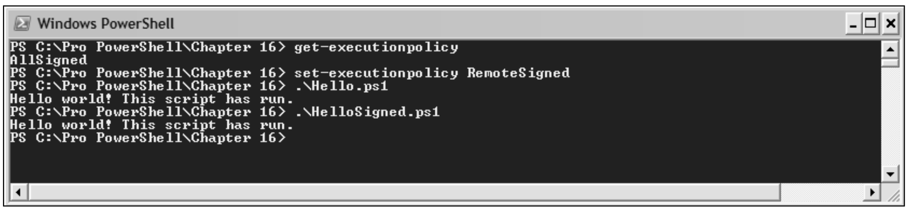


Figure 16-6

If you set execution policy to `Unrestricted`, both locally created scripts (unsigned and signed) run without prompting the user for a decision. Since both `RemoteSigned` and `Unrestricted` allow locally created scripts to run, they are suitable for a developer machine where multiple executions of a script during development and debugging are likely.

The Certificate Namespace

The certificate namespace contains certificate objects whose names are unique in that namespace. The certificate namespace is exposed as the `cert` drive.

To navigate to the `cert` drive from any other Windows PowerShell drive, simply type:

```
cd cert:
```

As with other PowerShell drives, ensure you include the final colon character when specifying the desired drive or an error message will be displayed. Once in the `cert:` drive, you can, as in other Windows PowerShell drives, use the `get-childitem` cmdlet to explore the content of the drive.

The cert drive has two folders inside it, `LocalMachine` and `CurrentUser`. To see those, type the following command when `Cert:\` is the current location:

```
get-childitem
```

Switch to the CurrentLocation folder using this command:

```
set-location CurrentUser
```

or:

```
cd CurrentUser
```

then type:

```
get-childitem
```

or:

```
dir
```

and you will see the second level folders shown in Figure 16-7. Notice the My folder, which is where the certificate you will create later in this chapter will be stored.

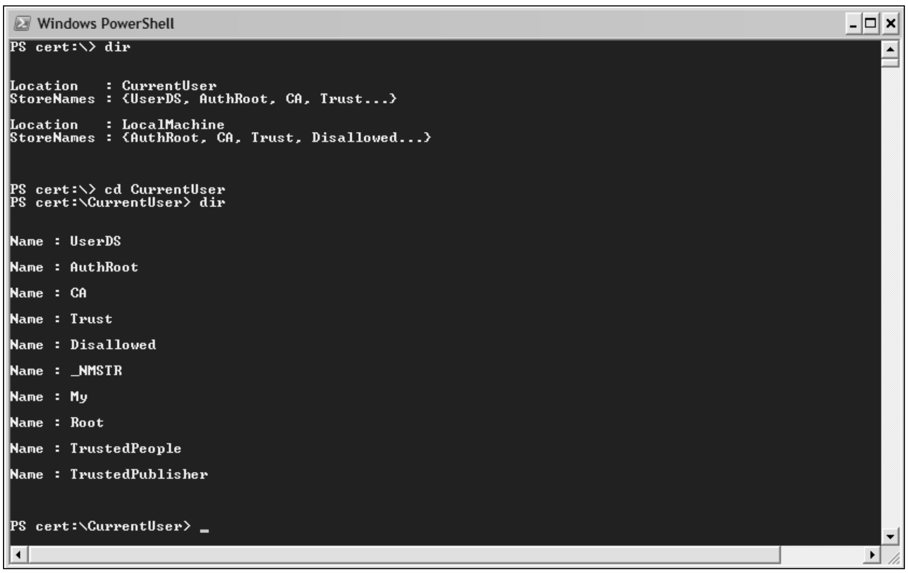


Figure 16-7

When working with the `cert` *drive from the Windows PowerShell command line, be aware that it does not always immediately update with newly created certificates. Opening a new Windows PowerShell command shell causes it to recognize the newly created certificate.*

Signed Scripts

Windows PowerShell provides two script-signing cmdlets, the `set-authenticodesignature` and `get-authenticodesignature` cmdlets. These enable you to sign scripts and to examine the signature of a script, respectively.

Creating a Certificate

To use the `set-authenticodesignature` and `get-authenticodesignature` cmdlets, you need to be able to create code-signing certificates on the machine. If you have access to a corporate code-signing certificate, you may prefer to use that to follow through this example. If you want to distribute signed scripts later, you will need a commercial code-signing certificate. The instructions provided here are based on the `makecert.exe` utility included in the .NET Framework 2.0 SDK, which comes with Visual Studio 2005.

Creating a certificate for Windows PowerShell using `makecert.exe` is a two-step process. First, navigate to the location in which you installed the `makecert.exe` utility and create a Windows PowerShell Local Certificate Root using the following command:

```
makecert -n "CN=Windows PowerShell Local Certificate Root" -a sha1 `
         -eku 1.3.6.1.5.5.7.3.3 -r -sv root.pvk root.cer `
         -ss Root -sr localMachine
```

You will be prompted for a password in a separate window. Assuming that you typed the command correctly, you will see a `Succeeded` message similar to the one shown in Figure 16-8.

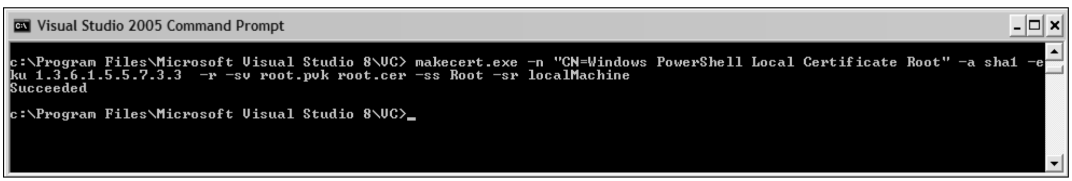


Figure 16-8

Next, you create a code-signing certificate. Use the following command to create a certificate named `Pro Windows PowerShell Test Certificate` in the `My` folder of the `CurrentUser` folder of the `cert` drive:

```
makecert -pe -n "CN=Pro Windows PowerShell Test Certificate" -ss MY -a sha1 `
         -eku 1.3.6.1.5.5.7.3.3 -iv root.pvk -ic root.cer
```

The usage of the switches for the `makecert.exe` utility is described in the .NET Framework 2.0 documentation. You will be prompted to enter the password you entered in the previous step.

Assuming that you have successfully created the certificate in the preceding command, navigate to the My folder, and you will be able to verify that it's there using the following command:

```
get-childitem * |
where-object {$_.Subject -match "CN=Pro Windows PowerShell"}
```

If you're using another name for your certificate, modify the regular expression in the second step of the pipeline accordingly. Figure 16-9 shows the information about the `Pro Windows PowerShell Test Certificate` certificate.

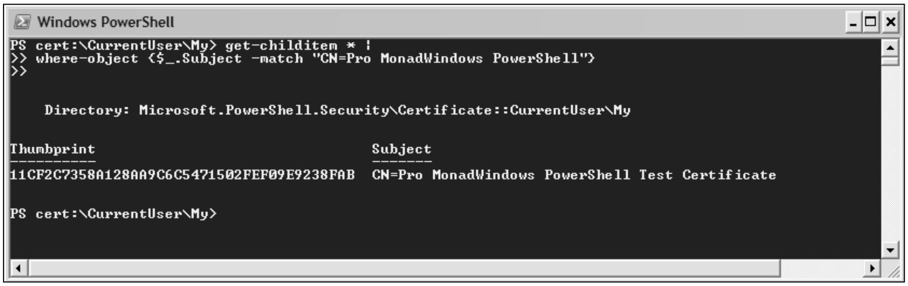


Figure 16-9

Now that you have a successfully created certificate, you are in a position to explore using the `set-authenticodesignature` and `get-authenticodesignnature` cmdlets.

The set-authenticodesignature Cmdlet

The `set-authenticodesignature` cmdlet places an authenticode signature in a Windows PowerShell script file. In addition to the common parameters (described in Chapter 6), the `set-authenticodesignature` cmdlet supports the following parameters:

- `FilePath` — Specifies the location of the script file to be signed.
- `Certificate` — Specifies the certificate to use to sign the script.
- `IncludeChain` — Specifies how much of the certificate trust chain to include in the signature.
- `TimeStampServer` — Specifies the URL of a timestamp server.
- `Whatif` — The user is informed what would have happened, but no change is made.
- `Confirm` — The user is requested to confirm the action.

The following commands show how to sign the `HelloSigned2.ps1` script with the `Pro Windows PowerShell Test Certificate` certificate.

Create the (for the moment) unsigned script file using this command:

```
set-content HelloSigned2.ps1 'write-host "Hello world! This script has run."'
```

I know that it is at `element [1]` in the array of code-signing certificates available in the `cert:\CurrentUser\My` folder. If your certificate is in a different location, amend the commands accordingly.

```
$file = "C:\Pro PowerShell\Chapter 16\HelloSigned2.ps1"
$cert = @(get-childitem cert:\CurrentUser\My\ -codesign)[1]
set-authenticodesignature -Path $file  -Certificate $cert
```

Figure 16-10 shows the script file `SignedHello2.ps1` being successfully signed. Notice the change in file length after the file has been signed.

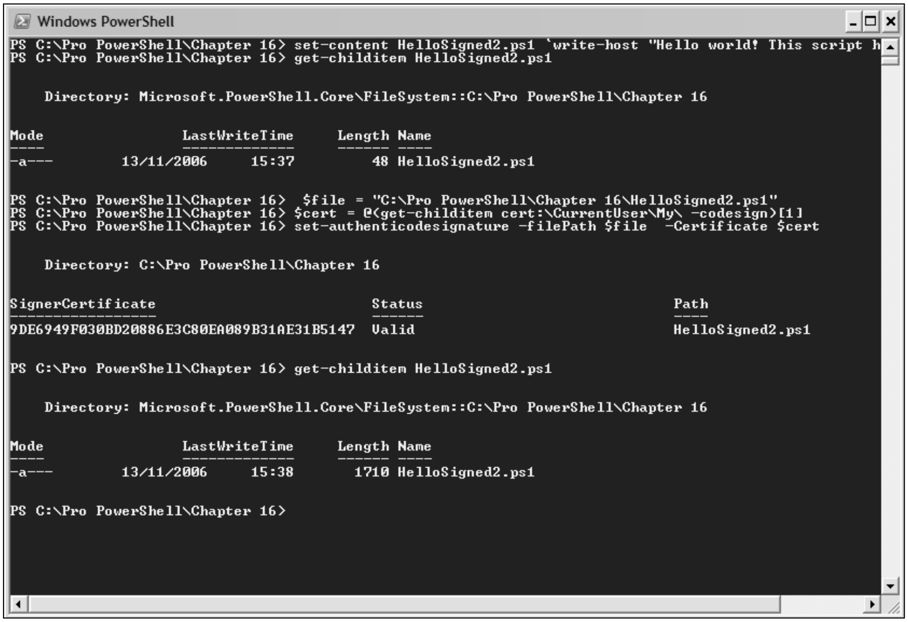


Figure 16-10

Once a script has been signed the signature information is encoded in multiple Windows PowerShell comments at the end of the script, as shown in Figure 16-11.

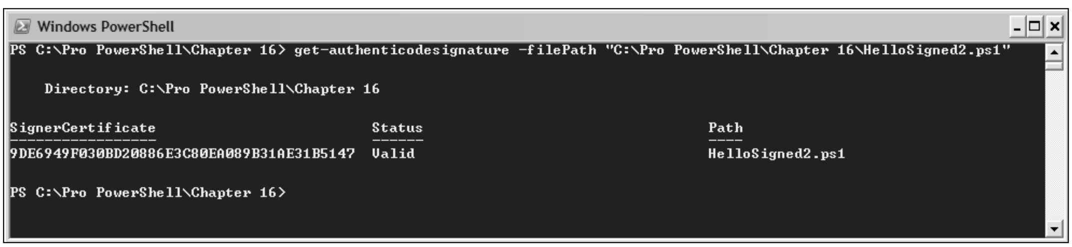


Figure 16-11

The get-authenticodesignature Cmdlet

The `get-authenticodesignature` cmdlet retrieves a signature object corresponding to the signature at the end of the script file.

In addition to the common parameters, the `get-authenticodesignature` cmdlet supports a `filePath` parameter, which specifies the path to the script file whose signature is the focus of interest.

To get the signature information for the script `HelloSigned2.ps1`, use the following command (assuming that the file is located in the directory `C:\Pro PowerShell\Chapter 16`):

```
get-authenticodesignature -filePath "C:\Pro PowerShell\Chapter 16\HelloSigned2.ps1"
```

As you can see in Figure 16-12, information about the certificate used to sign the script is retrieved.

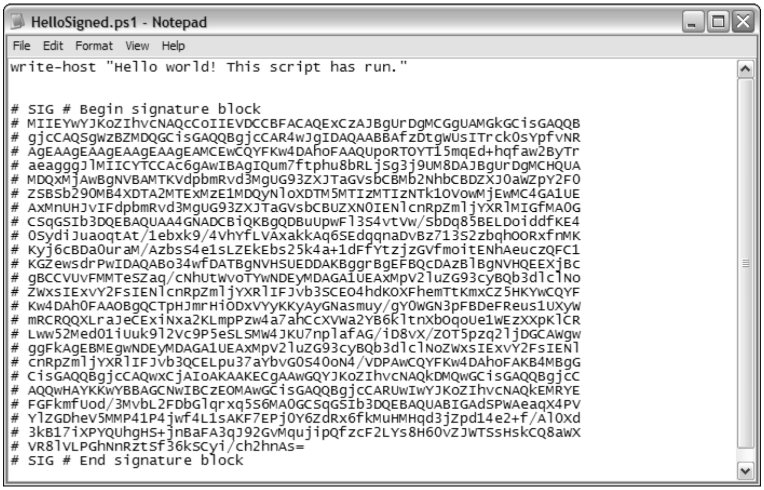


Figure 16-12

Summary

Windows PowerShell is designed to be “Secure by Default.” The default settings provide protection against executing scripts inadvertently or allowing malicious profile files to execute.

The default execution policy is `Restricted`. Once you make a decision to open up script execution, you need to be aware of the possible dangers of executing scripts whose source is untrusted or whose content is not understood.

The responsibility for which scripts to run is yours!

You can find the current execution policy using the `get-executionpolicy` cmdlet. You can modify the current execution policy by using the `set-executionpolicy` cmdlet. The `set-executionpolicy` cmdlet supports the following values for the execution policy:

- `Restricted` (the default)
- `AllSigned`
- `RemoteSigned`
- `Unrestricted`

To sign a PowerShell script, you need a code-signing certificate. I demonstrated how to sign a script using the `makecert.exe` utility and the `set-authenticodesignature` cmdlet. You can use the `get-authenticodesignature` cmdlet to retrieve signature information from a signed cmdlet.

17

Working with Errors and Exceptions

One of the realities of life when you're working with computers is that, no matter how careful you are, something is going to go wrong somewhere — and things will go wrong sufficiently often that it's important to recognize and prepare for the possibility. For this reason, it makes sense to provide Windows PowerShell with functionality to monitor and respond intelligently to error conditions. And that's what the Windows PowerShell team have done.

This chapter introduces the way Windows PowerShell treats errors and shows you how to retrieve information about errors and how you can change the way Windows PowerShell responds to errors and exceptions.

Errors in PowerShell

In some Windows PowerShell material a distinction is drawn, conceptually and practically, between *terminating errors* and *nonterminating errors*. Information about both types of errors is stored in the `$Error` variable, which is described in the next section.

The dividing line between terminating and nonterminating errors is a little fuzzy and depends in part on the perceptions of the author of a PowerShell or custom cmdlet. When the cmdlet author considers that a terminating error is appropriate, then the `ThrowTerminatingError()` method of the `System.Management.Automation.Cmdlet` class is called. If the cmdlet author deems that a nonterminating error is appropriate, then the `WriteError()` method of the `System.Management.Automation.Cmdlet` class is called. For cmdlets that depend on the presence of the PowerShell runtime, the corresponding `ThrowTerminatingError()` and `WriteError()` methods of the `System.Management.Automation.PSCmdlet` class are used for terminating and nonterminating errors, respectively.

A terminating error, broadly, has the following characteristics:

- It occurs when the cmdlet author considers that processing of the current object or further objects cannot be carried out in specified circumstances.
- It is used when the cmdlet author does not want processing of the current object or further objects to be carried out in specified circumstances
- `$Error[0]` contains information about the terminating error, if it is the most recent error of either kind.
- Processing stops.
- The terminating error can be caught by using the `Trap` statement (which I introduce later in this chapter).

The characteristics of a nonterminating error, broadly, are as follows:

- `$Error[0]` contains information about the error.
- Processing continues.

The `$?` variable contains the value `True` if the preceding statement executed successfully. It contains the value `False` if there was a terminating error in the preceding statement.

The following script, `ForLoop.ps1,` illustrates what happens to the `$?` variable when an error occurs. In this example, script the error is caused by an attempted division by zero.

```
write-host '$?'" is $? before the for loop."
for ($i = 5; $i -gt -5;$i--)
{write-host '$i'"=$i"
1/$i
write-host '$?'"=$?"
}
write-host '$?'"=$?"
```

The script uses a `for` loop to decrement a variable, `$i`. In each iteration of the loop, `$i is decremented.` The integer 1 is divided by `$i`, and the value of the variables `$?` and `$i` are displayed as the value of `$i` is decremented. When the value of the `$i` variable is zero, the division becomes 1/0, which creates an error. Notice in Figure 17-1 that the value of `$?` immediately after the error is `False`, whereas it had previously been `True`.

```
Attempted to divide by zero.
At C:\Pro PowerShell\Chapter 17\ForLoop.ps1:6 char:3
+ 1/$ <<<< i
$? =False
```

However, execution of the script continues for the values `-1` down to `-5`.

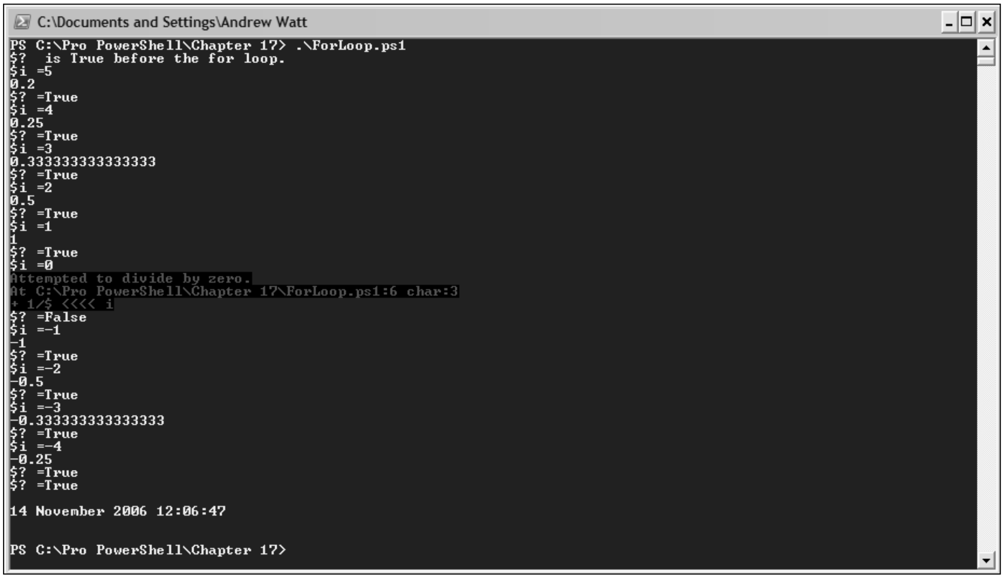


Figure 17-1

$Error

`$Error` is a Windows PowerShell system variable that contains an array of information about recent errors. Individual errors are accessed by using array notation, for example, the command

```
$Error[0]
```

returns element `0` of the `Error` array, which represents the error message for the most recent error.

When the Windows PowerShell shell is first opened the command `Error[0]` returns nothing to the console, since there have been no errors in the PowerShell session (assuming successful startup) and the Windows PowerShell prompt simply moves on to the next line. However, if you have run a statement that produces an error, for example

15/0

an error message is displayed, as shown in Figure 17-2, typing the following command returns the error message of the most recent error:

```
$Error[0]
```

Figure 17-2

If you run another command that is known to produce an error, for example, attempting to find a file

```
get-childitem Fred.txt
```

in a folder that doesn't have such a file, the divide by zero error is now stored in `$Error[1]` and the new error message about the file not being found is now returned by `$Error[0]`. You can demonstrate this by typing the following commands:

```
$Error[0]
get-childitem Fred.txt
$Error[1]
$Error[0]
```

The results of running the preceding commands are shown in Figure 17-3.

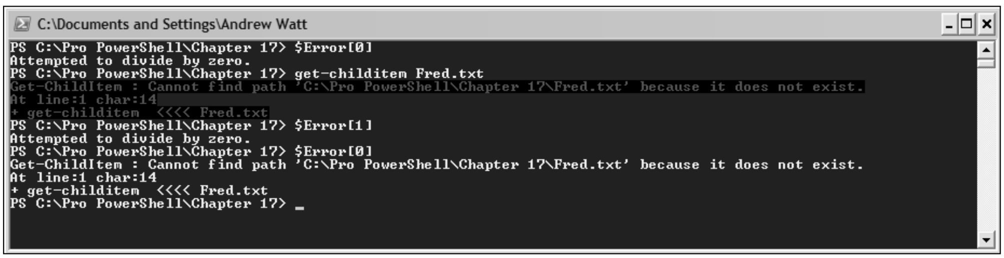


Figure 17-3

If you then run another command that generates a further error, for example

```
nota-cmdlet
```

which attempts to run a nonexistent cmdlet, `$Error[2]` now contains the information about the divide by zero error, and `$Error[1]` now contains the information about the `Cannot find path` error, as you can see in Figure 17-4.

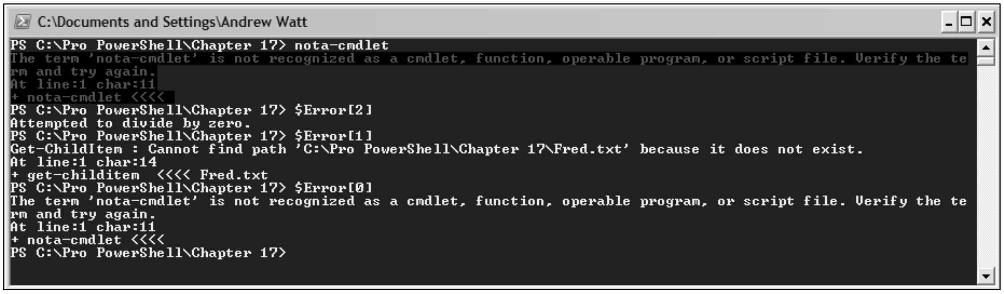


Figure 17-4

Information about each error, as with everything in Windows PowerShell, is stored in an object. To find out about the members of an error, use this command:

```
$Error[0] |
get-member
```

As you can see in Figure 17-5, there are several methods and six properties. Not all properties of an `ErrorRecord` object have values that can be displayed for each error that has occurred.

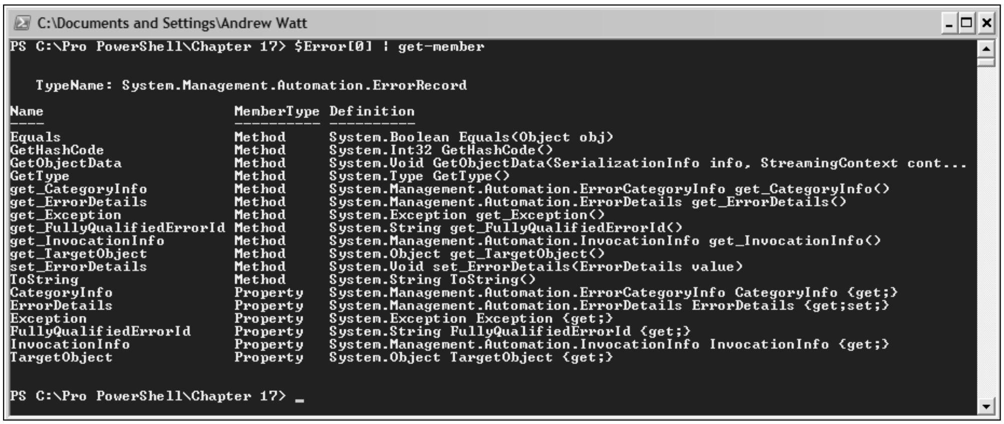


Figure 17-5

Not all elements of the `$Error` listarray are of the same type, nor do they have the same members. You can demonstrate that by running the following commands (which assumes the contents of `$Error[2]` and `$Error[1]` shown in Figure 17-4).

```
$Error[1] |
get-member -memberType Property
$Error[2] |
get-member -memberType Property
```

Figure 17-6 shows the results of executing the preceding commands.

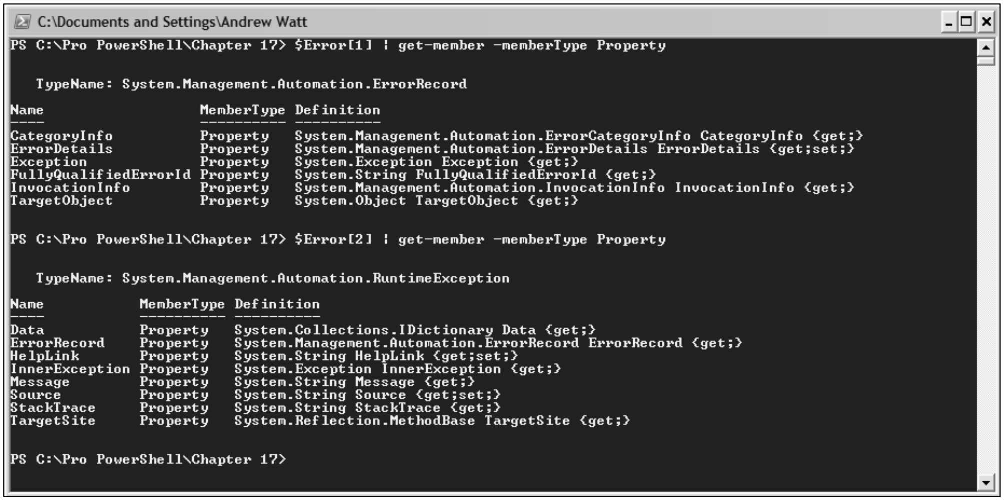


Figure 17-6

Just after running the script `ForLoop.ps1` used earlier in this chapter, `$Error[0]` contains information about a divide by zero error. As you can see in Figure 17-7, the `Exception` property holds the error message and the `InvocationInfo` property contains information associating the occurrence of the error with the script `ForLoop.ps1`. To see that information, execute these commands:

```
$Error[0].Exception |
format-list *
$Error[0].InvocationInfo |
format-list *
```

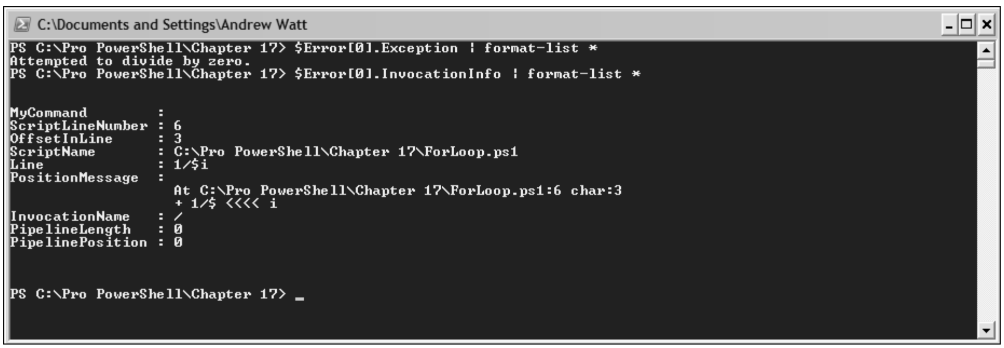
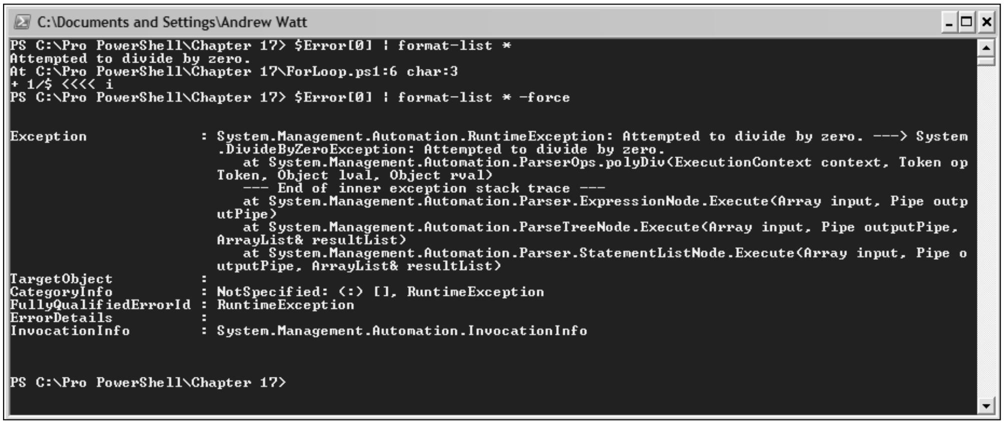


Figure 17-7

You can display similar but more succinct error information using the command:

```
$Error[0] |
format-list *
```

Using the `force` parameter of the `format-list` cmdlet, you can display all six properties, as shown in Figure 17-8.



```
C:\Documents and Settings\Andrew Watt
PS C:\Pro PowerShell\Chapter 17> $Error[0] | format-list *
Attempted to divide by zero.
At C:\Pro PowerShell\Chapter 17\ForLoop.ps1:6 char:3
+ 1/$ <<<< i
PS C:\Pro PowerShell\Chapter 17> $Error[0] | format-list * -force


Exception             : System.Management.Automation.RuntimeException: Attempted to divide by zero. ---> System
                        .DivideByZeroException: Attempted to divide by zero.
                           at System.Management.Automation.ParserOps.polyDiv(ExecutionContext context, Token op
                        Token, Object lval, Object rval)
                           --- End of inner exception stack trace ---
                           at System.Management.Automation.Parser.ExpressionNode.Execute(Array input, Pipe outp
                        utPipe)
                           at System.Management.Automation.ParseTreeNode.Execute(Array input, Pipe outputPipe,
                        ArrayList& resultList)
                           at System.Management.Automation.Parser.StatementListNode.Execute(Array input, Pipe o
                        utputPipe, ArrayList& resultList)
TargetObject          :
CategoryInfo          : NotSpecified: (:) [], RuntimeException
FullyQualifiedErrorId : RuntimeException
ErrorDetails          :
InvocationInfo        : System.Management.Automation.InvocationInfo


PS C:\Pro PowerShell\Chapter 17>
```

Figure 17-8

The number of errors contained in `$Error` is defined in the `$MaximumErrorCount` variable. By default, 256 errors are stored before older errors begin to be discarded. You can increase the value displayed by `$MaximumErrorCount` by using a command like the following:

```
$MaximumErrorCount = 2000
```

PowerShell prevents you decreasing the value of `$MaximumErrorCount` below 256. For example, the command

```
$MaximumErrorCount = 255
```

produces the following error message:

```
Cannot validate because of invalid value (255) for variable MaximumErrorCount.
At line:1 char:19
+ $MaximumErrorCount  <<<< = 255
```

You can clear the content of `$Error` by using the following command:

```
$Error.Clear()
```

You can delete a range of values in `$Error` by using the `RemoveRange()` method. The arguments are the starting index and the number of elements to be removed. For example, the following command removes 10 values starting at index 3:

```
$Error.RemoveRange(3,10)
```

To clear `$Error` using the `RemoveRange()` method, you can use this command:

```
$Error.RemoveRange(0,$Error.Count)
```

although `$Error.Clear()` is simpler.

Using Error-Related variables

Windows PowerShell provides several variables that are relevant to how you work with errors. To view error-related variables execute this command:

```
get-variable *error*
```

Figure 17-9 shows the error-related variables.

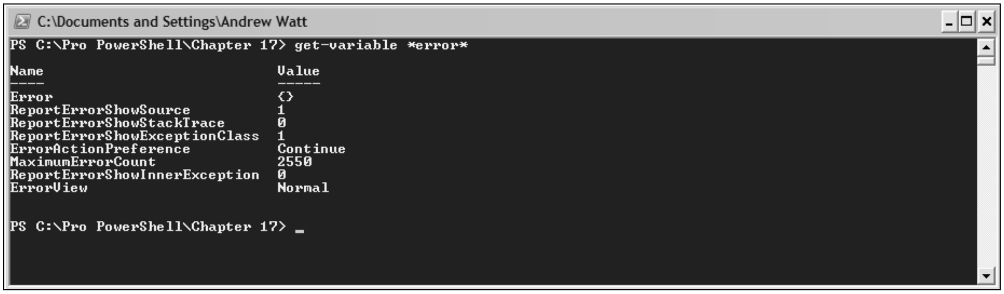


Figure 17-9

To view further information on the error-related variables, use this command:

```
get-variable *error* |
format-list
```

Figure 17-10 shows the results of executing the preceding command.

Notice that the `Constant` option applies to `$Error` (but none of the others), which means that you can't delete `$Error`. If you attempt to delete it using the following command

```
remove-item variable:Error
```

the following error message is displayed:

```
Remove-Item : Cannot remove variable Error because it is constant or read-only. If
the variable is read-only, try the operation again specifying the Force option.
At line:1 char:12
+ remove-item  <<<< variable:Error
```

In principle, you can delete other error-related variables, although I can't see a good reason why you would benefit from doing that.

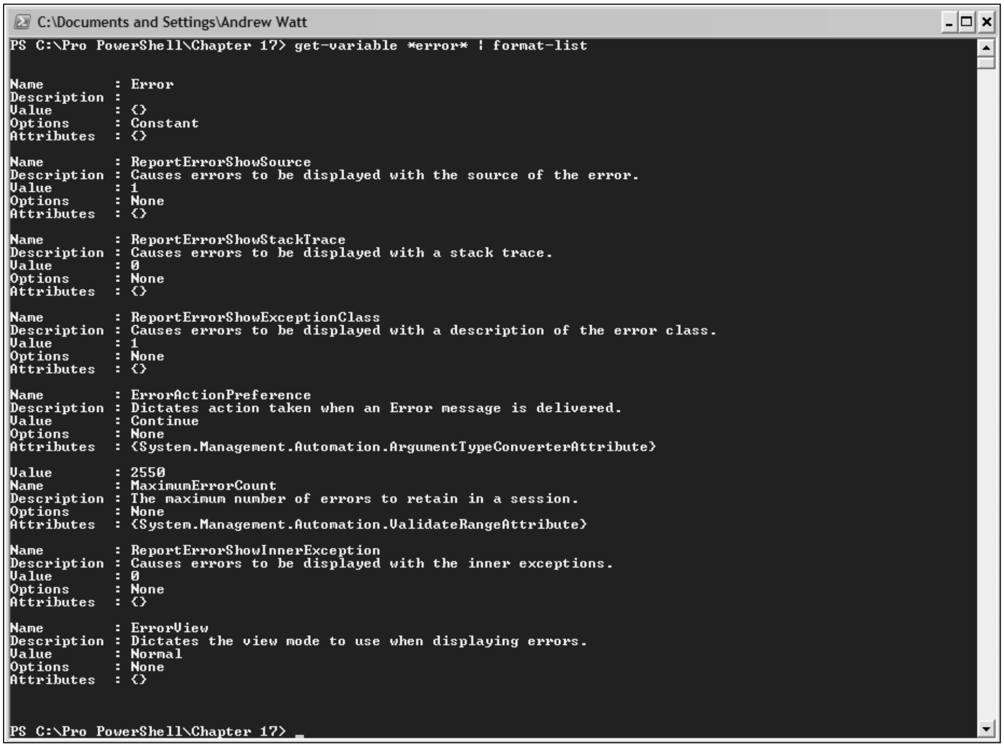


Figure 17-10

Using the $ErrorView variable

PowerShell supports two views on to the elements of `$Error`, called `NormalView` and `CategoryView`. Not surprisingly, the default view is `Normal`. The `CategoryView` option is intended to provide succinct, highly structured informative error information. This would be useful, for example, in a high-volume setting where the likely causes of error are well known.

To see the current setting of the `$ErrorView` variable, type this command:

```
$ErrorView
```

The supported values are `NormalView` and `CategoryView`.

The `CategoryView` view is generally succinct and well structured. Compare the results in Figure 17-11 of executing the following to find a nonexistent file:

```
get-childitem Fred.txt
```

Figure 17-11

Sometimes the information displayed in `CategoryView` view is less informative than that in the `Normal` view. Compare the information displayed when executing the following command, as shown in Figure 7-12:

```
1 / 0
```

Figure 17-12

In `NormalView`, the following error message is displayed:

```
Attempted to divide by zero.
At line:1 char:3
+ 1/0 <<<<
```

In `CategoryView`, view the error message is more succinct but not particularly informative:

```
NotSpecified: (:) [], ParentContainsErrorRecordException
```

Using the $ErrorActionPreference variable

The `$ErrorActionPreference` variable specifies the action to take in response to an error occurring. The following values are supported:

- ❑ `SilentlyContinue` — Don't display an error message continue to execute subsequent commands.
- ❑ `Continue` — Display any error message and attempt to continue execution of subsequence commands.
- ❑ `Inquire` — Prompts the user whether to continue or terminate the action
- ❑ `Stop` — Terminate the action with error.

Set the `$ErrorActionPreference` variable to `SilentlyContinue` by using this command:

```
$ErrorActionPreference = "SilentlyContinue"
```

As you can see in Figure 17-13, the `ForLoop.ps1` script runs to completion without displaying any error message. The error message is available in `$Error[0]` if you want to see it.

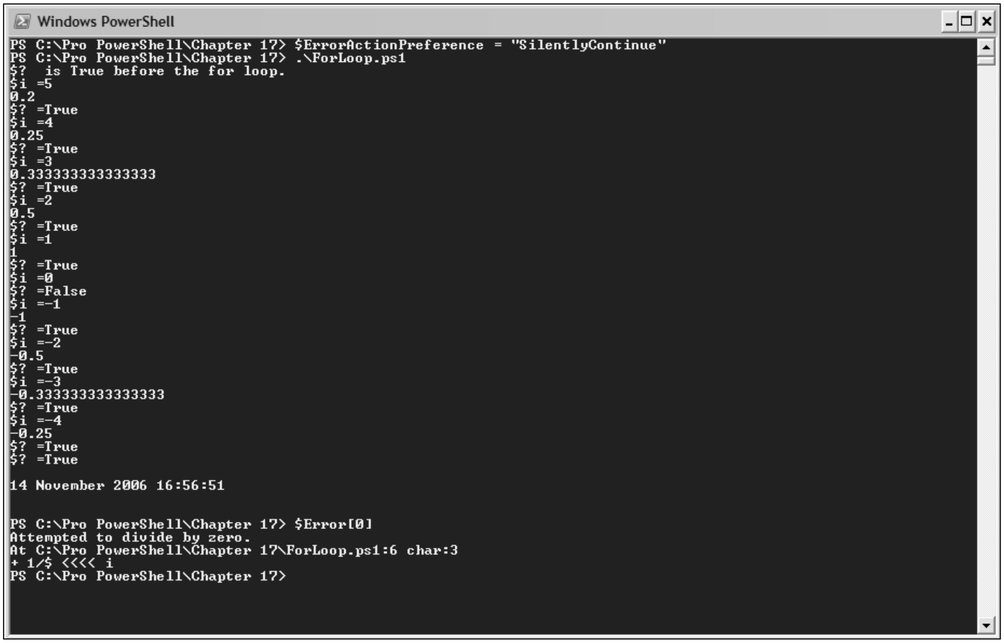


Figure 17-13

The default value of `$ErrorActionPreference` is `Continue`. When that is the setting, the appearance when running `ForLoop.ps1` is the same as that shown in Figure 17-1.

When you set the value of `$ErrorActionPreference` to `Inquire`, execution stops when the error occurs, the error is described, and the user is prompted to decide what to do next:

```
Action to take for this exception:
Attempted to divide by zero.
[C] Continue  [I] Silent Continue  [B] Break  [S] Suspend  [?] Help (default is
"C"):
```

Figure 17-14 shows the results of executing `ForLoop.ps1` with `$ErrorActionPreference` set to `Inquire`.

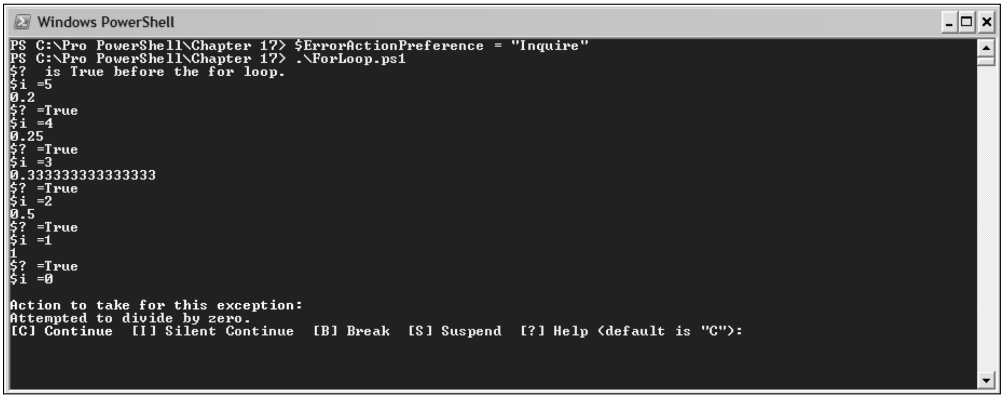


Figure 17-14

When the value of `$ErrorActionPreference` is set to `Stop`, the error message is displayed, and execution of the script stops, as shown in Figure 17-15.

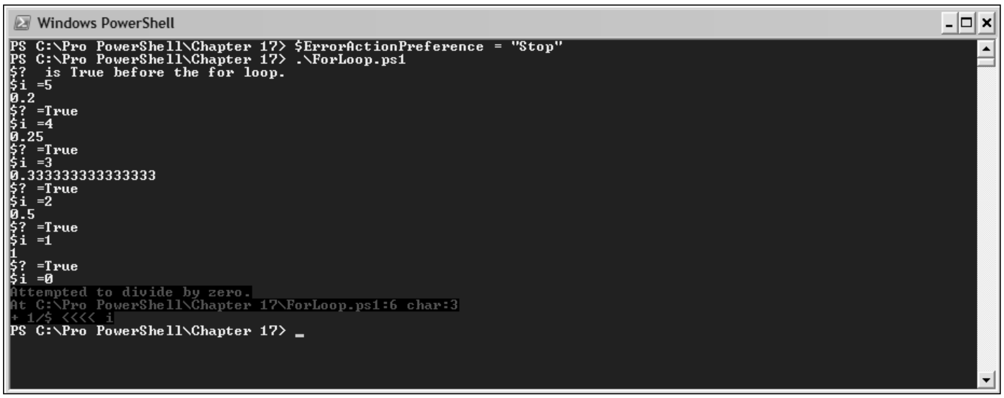


Figure 17-15

Trap Statement

The `trap` statement allows you to take control of what happens when an error occurs.

Don't attempt to use the trap statement to trap nonterminating errors. It doesn't work for those.

The script `TrappedForLoop.ps1` writes a custom message to the console when an exception occurs and specifies that execution is to continue:

```
for ($i = 5;  $i -gt -2; $i--)
{
trap {write-host "This is a custom error message.";continue}
```

```
write-host '$i'"=$i"
1/$i
write-host '$?'"=$?"
}
write-host "This statement comes after the for loop."
```

As the value of `$i` is decremented, eventually the statement `1/$i` becomes `1/0`, which causes an error. The error is trapped and the custom error message is displayed, as you can see in Figure 17-16. Even when the `$ErrorActionPreference` is set to `Stop`, execution of the `for` loop continues, as does execution of statements following the `for` loop.

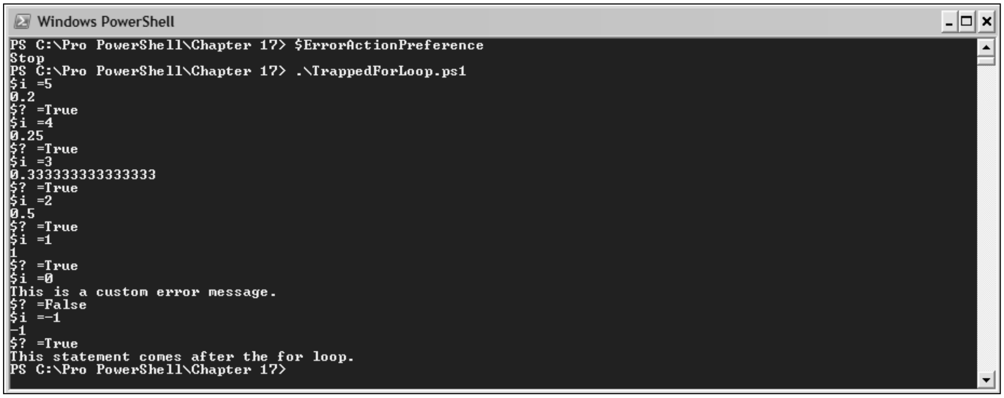


Figure 17-16

The following script, `BlockedWrite.ps1`, writes sample text to four files (using redirection), sets the file `Test3.txt` to read-only, then attempts to append further text to `Test3.txt`.

```
Trap {write-host "You can't write to a read-only file."}
"This is file 1" > Test1.txt
"This is file 2" > Test2.txt
"This is file 3" > Test3.txt
"This is file 4" > Test4.txt
attrib +r Test3.txt
"Add to file 3" >> Test3.txt
```

Execute the script using the following command:

```
.\BlockedWrite.ps1
```

As you can see in Figure 17-17, the custom error message specified in the `trap` statement is displayed. The normal error message is also displayed following the custom error message. This is because the default behavior of a `Trap` statement is `Return`.

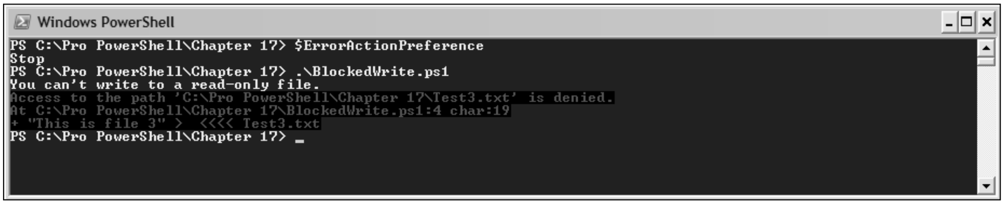


Figure 17-17

If you want to take any custom action and suppress the normal error message, use the `Continue` statement in the `trap` statement's script block. Remember to separate statements on a single line using a semicolon. The script (which includes the `Continue` statement) `BlockedWrite2.ps1` is shown below. Before running the script manually, delete the test files after making `Test3.txt` `read-write` using the following command:

```
attrib -r Test3.txt
```

If you don't delete the files (or set the file to read-write), you will see the error message twice when you execute the script — once for the fourth line of the script and once for the final line — since both attempts to write to `Test3.txt` fail because it was set earlier to read-only by the script `BlockedWrite.ps1`.

```
Trap {write-host "You can't write to a read-only file.";Continue}
"This is file 1" > Test1.txt
"This is file 2" > Test2.txt
"This is file 3" > Test3.txt
"This is file 4" > Test4.txt
attrib +r Test3.txt
"Add to file 3" >> Test3.txt
```

Figure 17-18 shows the result of running the script `BlockedWrite2.ps1`.

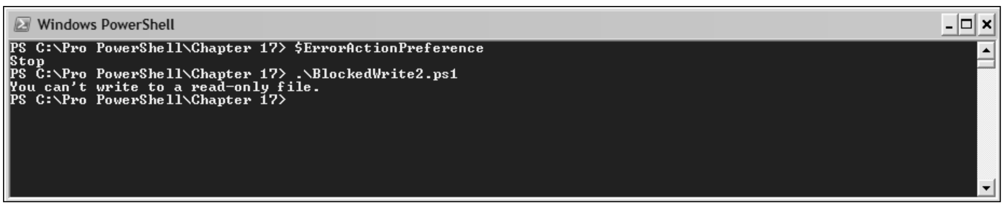


Figure 17-18

In the following two examples, don't delete `Test3.txt` before attempting to run either script. Both attempts to write to the file in each script should fail. The aim is to show the difference in behavior with multiple errors with a `continue` or `break` statement in the statement block of a `trap` statement.

In the following script, `BlockedWrite3.ps1`, I have added `write-host` statements to show you that the statement immediately before and immediately after the two attempts to write to `Test3.txt` have been reached.

```
Trap {write-host "You can't write to a read-only file.";Continue}
"This is file 1" > Test1.txt
"This is file 2" > Test2.txt
write-host "The first write to Test3.txt hasn't happened yet."
"This is file 3" > Test3.txt
write-host "The first write to Test3.txt is over."
"This is file 4" > Test4.txt
attrib +r Test3.txt
write-host "The append to Test3.txt is about to be attempted."
"Add to file 3" >> Test3.txt
write-host "The append to Test3.txt is over."
```

Notice the `Continue` statement in the `Trap` statement. Figure 17-19 shows the results. As you can see, both writes to `Test3.txt` have been attempted, as demonstrated by the execution of the `write-host` statements immediately before and after each of the attempted writes.

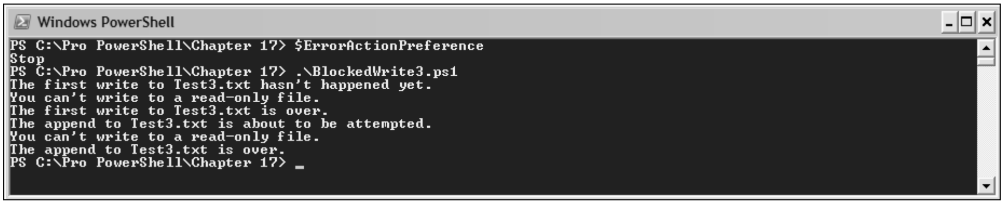


Figure 17-19

However, if you modify the `Trap` statement so that `Break` replaces `Continue`, execution stops after the first error. This is `BlockedWrite4.ps1`, which includes the `Break` statement in the `Trap` statement.

```
Trap {write-host "You can't write to a read-only file.";Break}
"This is file 1" > Test1.txt
"This is file 2" > Test2.txt
write-host "The first write to Test3.txt hasn't happened yet."
"This is file 3" > Test3.txt
write-host "The first write to Test3.txt is over."
"This is file 4" > Test4.txt
attrib +r Test3.txt
write-host "The append to Test3.txt is about to be attempted."
"Add to file 3" >> Test3.txt
write-host "The append to Test3.txt is over."
```

As you can see in Figure 17-20, when `Break` is used in the `Trap` statement the code in the `Trap` statement's script block is executed (in this case displaying the custom error message), the normal error message is displayed, then execution stops. The statement

```
write-host "The first write to Test3.txt is over."
```

is never executed, nor are any of the later statements in the script.

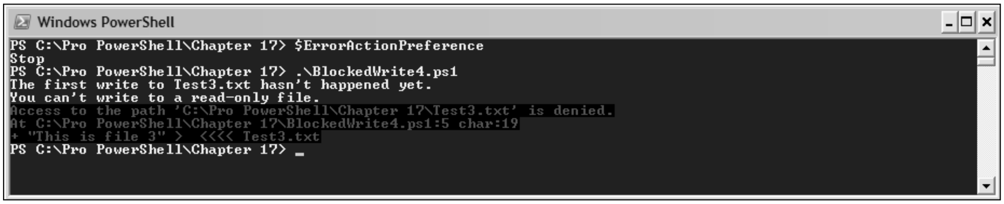


Figure 17-20

The `trap` statement also supports an option so that the statement's script block is executed only in response to a specific type of error.

The `write-error` statement allows you to output a customized error message when an error occurs. The type of error also changes when you use the `write-error` cmdlet.

Run the `TrappedForLoop.ps1` script again. Then run this command:

```
$Error[0] |
format-list * -force
```

Figure 17-21 shows the results. Notice that the exception is a `System.Management.Automation.RuntimeException`.

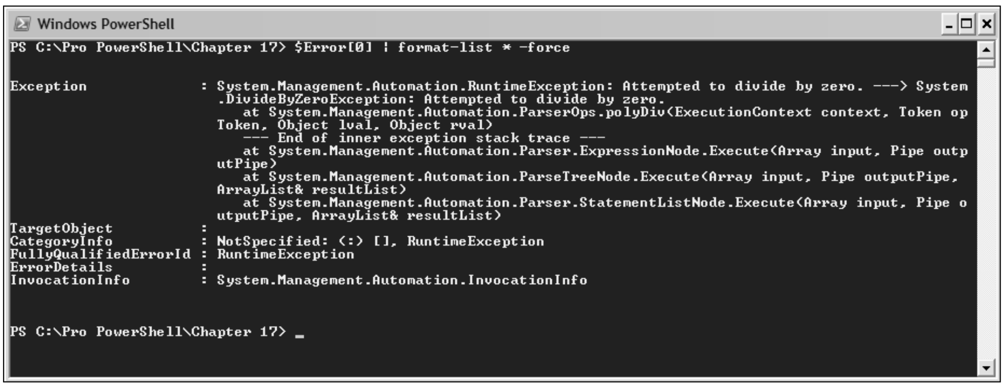


Figure 17-21

Next run `TrappedForLoop2.ps1`, where a write-error statement has been added to the statement block of the `trap` statement:

```
for ($i = 5;  $i -gt -2; $i--)
{
trap {write-host "This is a custom error message."; write-error "You attempted

to divide by zero!!!";continue}
write-host '$i'"=$i"
1/$i
```

```
write-host '$?'"=$?"
}
write-host "This statement comes after the for loop."
```

Run the script:

```
.\TrappedForLoop2.ps1
```

then run this command:

```
$Error[0] | format-list * -force
```

Figure 17-22 shows the results. Notice that the exception is now a `Microsoft.PowerShell.Commands.WriteErrorException` and that the custom error message specified in the write-error statement is displayed with the same appearance as a built-in error.

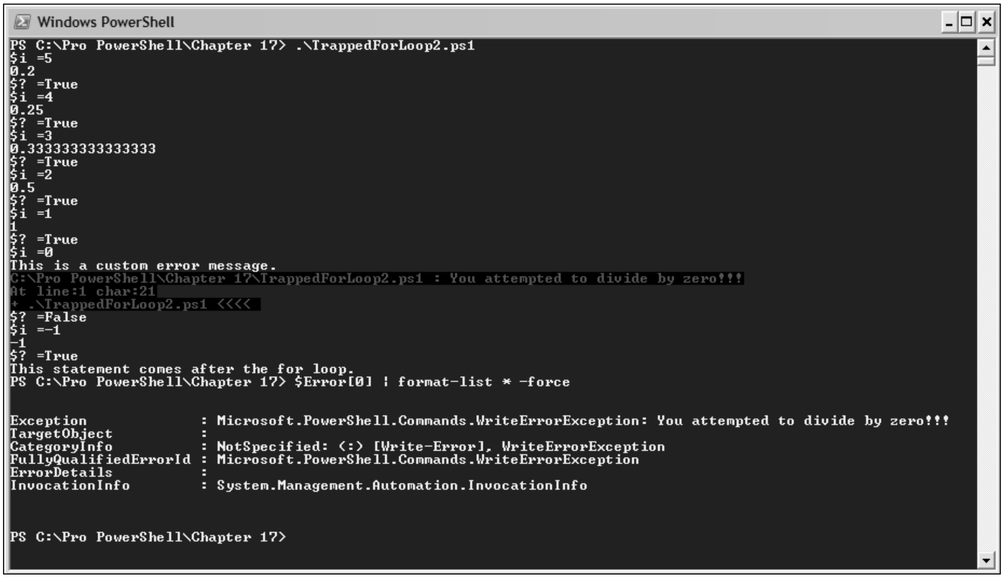


Figure 17-22

Using Common Parameters

Many cmdlets support the common parameters `ErrorAction` and `ErrorVariable`. These are particularly relevant to the situation where errors can occur or are likely to occur.

Using the ErrorAction Parameter

The `-errorAction` parameter specifies the error action for a cmdlet. It overrides the value set in the `$ErrorActionPreference` variable.

To show how you can use the `ErrorAction` parameter, enter these commands with your current working folder set to one that does not contain a file called `Fred.txt`.

First confirm the value of `$ErrorActionPreference` by running the following command:

```
$ErrorActionPreference
```

Then run each of the following commands:

```
get-childitem Fred.txt -ErrorAction SilentlyContinue ; write-host "We got here"
get-childitem Fred.txt -ErrorAction Continue; write-host "We got here"
get-childitem Fred.txt -ErrorAction Stop; write-host "We got here"
get-childitem Fred.txt -ErrorAction Inquire; write-host "We got here"
```

The commands use the four possible enumerated values for the `ErrorAction` parameter — `SilentlyContinue`, `Continue`, `Stop`, and `Inquire`. Figure 17-23 shows the results with blank lines added to help readability.

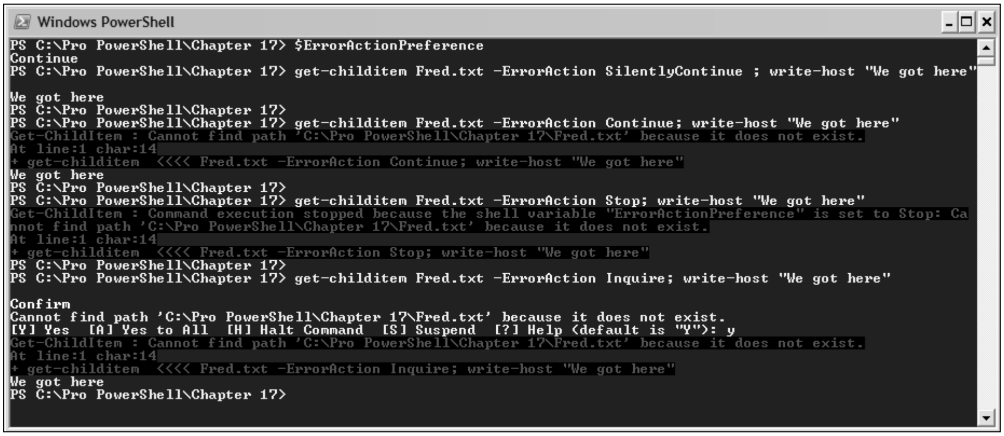


Figure 17-23

Notice that the value of `$ErrorActionPreference` is `Continue`.

With the `ErrorAction` parameter set to `SilentlyContinue`, the error message is not displayed and execution continues to the `write-host` statement. With the `ErrorAction` parameter set to `Continue`, the error message is displayed and execution continues to the `write-host` statement. With the value of the `ErrorAction` parameter set to `Stop` execution stops, a message is displayed about stopping, the usual error message is then displayed, but execution of the `write-host` statement does not take place. With the value of the `ErrorAction` parameter set to `Inquire`, the user is asked what to do. If the user selects the `Y (Yes)` option, then it is as if `ErrorAction` were set to `Continue`. If the user selects the `H (Halt)` option, then execution is halted, a message is displayed, which is a little different from the message with `ErrorAction`, having the value of `Stop`. The `write-host` statement is not executed.

Using the ErrorVariable Parameter

The `ErrorVariable` parameter allows you to specify a variable to hold information about any error(s) relating to execution of a cmdlet.

The script `ErrorVariable.ps1` shows you simple usage of the `ErrorVariable` parameter:

```
$ErrorActionPreference = "SilentlyContinue"
$file = read-host "Enter a file name"
get-childitem $file -ErrorVariable myErrorVar

if ($myErrorVar.Count -gt 0)
{
 write-host "I couldn't find the file $file."
}
else
{
 write-host "`nI found the file $file."
}
$myErrorVar |format-list * -force
$ErrorActionPreference = "Continue"
```

To suppress any system error messages, I set the value of the `$ErrorActionPreference` to `SilentlyContinue` at the beginning of the script. I set it back to its default value of `Continue` when the script concludes.

The script asks the user to supply a filename which is then opened. The command

```
get-childitem $file -ErrorVariable myErrorVar
```

specifies that the error variable for the `get-childitem` cmdlet is `$myErrorVar`. Thus, if there is an error, Windows PowerShell sends the details to `$myerrorvar`.

Be careful not to include the `$` sign when specifying the name of the error action variable.

Figure 17-24 shows the results after entering the name of a nonexistent file, `Fred.txt`, and then the result after entering the name of a file that does exist, `Test3.txt`.

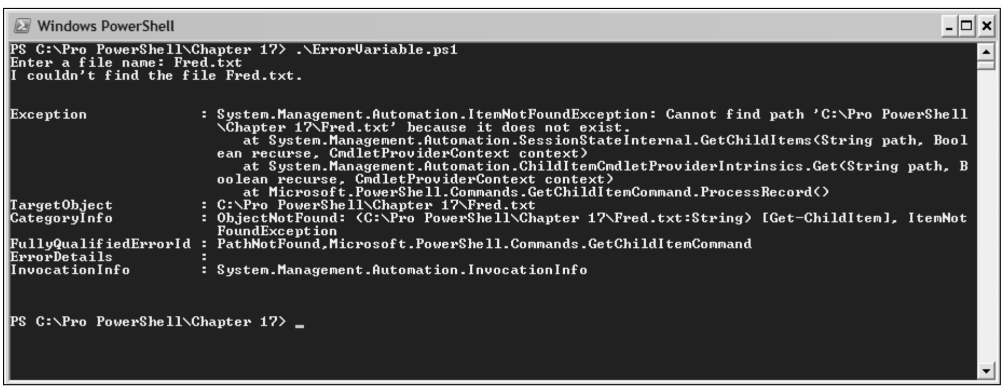


Figure 17-24

The write-error Cmdlet

The `write-error` cmdlet writes an error object and passes it to the pipeline. In addition to supporting the common parameters, the `write-error` cmdlet supports the following parameters:

- `Message` — Text that describes the error. Can be used in place of the `Exception` and `ErrorRecord` parameters. A required parameter.
- `Category` that — The category of error that the error is associated with. An optional parameter.
- `ErrorId` — The error ID associated with the error. An optional parameter.
- `TargetObject` — The object associated with the error. An optional parameter.
- `RecommendedAction` — The action recommended in response to the error. An optional parameter.
- `CategoryActivity` — A description of the activity that overrides the `ErrorCategoryInfo` default. An optional parameter.
- `CategoryReason` — A text description of the reason to override the `ErrorCategoryInfo` default. An optional parameter.
- `CategoryTargetName` — The Target Name to override the `ErrorCategoryInfo` default. An optional parameter.
- `CategoryTargetType` — The Target Type to override the `ErrorCategoryInfo` default. An optional parameter.
- `Exception` — The type of the error's exception. If used instead of the `Message` and `ErrorRecord` parameters, it is a positional parameter in position 1.
- `ErrorRecord` — An error record containing information about the error. If used instead of the `Message` and `Exception` parameters it is a positional parameter in position 1.

The permitted values of the `Category` parameter are listed here:

- `NotSpecified` — An error has occurred which isn't appropriate for another category.
- `CloseError` — An error that occurs when closing.
- `DeadlockDetected` — A deadlock has been detected.
- `DeviceError` — A device has reported an error.
- `FromStdError` — An error has been reported to STDERR.
- `InvalidArgument` — An invalid argument has been specified.
- `InvalidData` — An invalid type has been specified.
- `InvalidOperation` — An invalid operation has been requested.
- `InvalidResult` — An invalid result has been returned.
- `InvalidType` — An invalid type has been specified.
- `MetadataError` — There is an error in metadata.

- `NotImplemented` — A referenced API has not been implemented.
- `NotInstalled` — An item has not been installed.
- `ObjectNotFound` — An object cannot be found.
- `OpenError` — An error that occurs when opening.
- `OperationStopped` — An operation has stopped.
- `OperationTimeout` — An operation has timed out.
- `ParserError` — An error has occurred during parsing.
- `PermissionDenied` — An operation has been attempted without adequate permissions.
- `ReadError` — An error that occurs when reading.
- `ResourceBusy` — `A resource is busy.`
- `ResourceExists` — `A resource already exists.`
- `ResourceUnavailable` — A resource is unavailable.
- `SecurityError` — `A security error has occurred.`
- `SyntaxError` — There is a syntax error in a command.
- `WriteError` — An occur that occurs when writing.

Summary

Errors in Windows PowerShell are stored in the `$Error` variable. The default size is 256 errors. You can increase the size of `$Error` if desired.

Errors can be described as terminating errors and nonterminating errors. The distinction is not clear-cut.

The `$ErrorActionPreference` variable allows you to specify a global preference for PowerShell behavior when an error occurs.

The `$Error`View variable allows you to specify two views of error information.

Two of the common parameters are relevant to errors:

- The `-errorAction` paramter overrides for an individual cmdlet the value in `$ErrorActionPreference`.
- The –errorVariable parameter allows you to store error information in a variable other than `$Error`.

The `write-error` cmdlet allows you to create custom error messages.

18

Debugging

Writing error-free code is the aspiration of pretty much every programmer. If you've spent any significant amount of time writing programs of any kind, you'll know that writing error-free code becomes increasingly difficult as the size of your code increases.

When you use Windows PowerShell on the command line, identifying many errors is simply a matter of spotting some slight syntax error. But Windows PowerShell is a scripting tool as well as a command line shell, so as with any significant programming language, you will need to carry out at least some debugging of your code when writing Windows PowerShell scripts. The longer and more complex your PowerShell scripts become, the more demanding it is to identify and fix all the errors that are present in them.

Debugging is the process of trying to identify and correct bugs in PowerShell scripts or commands. Often during initial development of a script you will observe undesired behavior of some kind. Spotting what is wrong can be easy or it can be hugely time-consuming and sometimes frustrating.

You might fail to spot some types of errors because you don't test edge conditions. Until users run your scripts in conditions you hadn't anticipated, the code seems to run correctly. When users bring an unanticipated combination of conditions to code execution, previously unknown errors may surface. I don't propose to explore those issues in depth in this chapter but will focus primarily on issues that are specific to the debugging of PowerShell and its scripts.

Handling Syntax Errors

Most people who write code introduce syntax errors from time to time. When you're learning a new language or switching frequently between languages, syntax errors can become frequent. Windows PowerShell gives you some support in interpreting the syntax errors you introduce but, as with many other languages, you can expect some error messages to be only marginally helpful at best.

Simple syntax errors, such as misspelling a cmdlet name, are easily dealt with. If you type:

```
write-hos "Hello world!"
```

instead of:

```
write-host "Hello world!"
```

you will receive an error message about the command not being recognized, as shown in Figure 18-1.

```
The term 'write-hos' is not recognized as a cmdlet, function, operable program, or
script file. Verify the term
 and try again.
At line:1 char:10
+ write-hos  <<<< "Hello world!"
```

Figure 18-1

You know you were trying to type a cmdlet name, so it's that part of the error message that you should focus on. The error message doesn't tell you what the correct cmdlet name is, but the error message points you in the right direction — taking a close look at the command you used. Similar issues arise if you mistype the name of a function or script.

At other times, you will receive the same error message but not due to the supposed failure to recognize a script as a script. The script file, `Hello.ps1`, shown below, is used in the next example.

```
write-host "Hello world!"
```

If you attempt to run the script from the current directory (in my case `C:\Pro PowerShell\Chapter 18`) you see an error message that the script file is not recognized as a script, as shown in Figure 18-2.

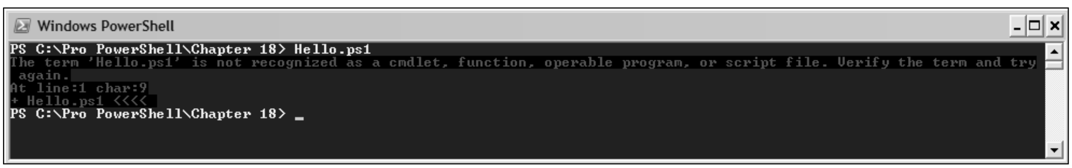


Figure 18-2

The error message is:

```
The term 'Hello.ps1' is not recognized as a cmdlet, function, operable program, or
script file. Verify the term and try
```

```
 again.
At line:1 char:9
+ Hello.ps1 <<<<
```

You know it is a script and there is nothing wrong with the (very simple) code in the script, as you can demonstrate by moving to the parent directory and executing the script using the following command:

```
&"Chapter 18\Hello.ps1"
```

The `&` in this context means execute what follows.

The problem is that, for perceived security reasons, a Windows PowerShell script in the current directory can be run by typing:

```
.\Hello.ps1
```

rather than typing:

```
Hello.ps1
```

Once you get beyond the simplest commands, the complexity increases and other error messages are potentially more difficult to work out. For example, in a `for` loop Windows PowerShell doesn't allow you to put a semicolon after the third component in the parentheses of the `for` statement. The script `ForLoopWithError.ps1` is shown here:

```
write-host '$?'" is $? before the for loop."
for ($i = 5; $i > 0; $i--;)
{

write-host '$i'"=$i"
1/$i
write-host '$?'"=$?"
}
write-host "This statement comes after the while loop."
write-host '$?'"=$?"
```

As you can see in Figure 18-3, the error message

```
Missing closing ')' after expression in 'for' statement.
At C:\Pro PowerShell\Chapter 18\ForLoopWithError.ps1:2 char:26
+ for ($i = 5; $i > 0; $i--; <<<< )
```

indicates that a parenthesis is missing, but inspection of the second line of the script shows the parentheses to be correctly paired.

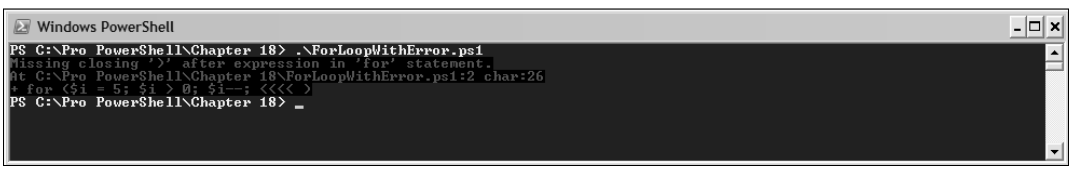


Figure 18-3

If you remove the third semicolon on the second line so that it reads

```
for ($i = 5; $i > 0; $i--)
```

and run the script `ForLoop.ps1`, shown below, it runs as designed.

```
write-host '$?'" is $? before the for loop."
for ($i = 5; $i > 0; $i--)
{

write-host '$i'"=$i"
1/$i
write-host '$?'"=$?"
}
write-host "This statement comes after the while loop."
write-host '$?'"=$?"
```

A particularly puzzling type of error is the error — or more precisely the unexpected behavior — for which you receive no error message at all. Take a look at the script `ForLoopWithUnreportedError.ps1` shown next, and see if you can spot the problem before reading the explanation.

```
write-host '$?'" is $? before the for loop.";
for ($i = 5; $i > 0; $i--)
{
write-host '$i'"=$i"
1/$i
write-host '$?'"=$?"
}
write-host "This statement comes after the for loop."
write-host '$?'"=$?"
```

Also, if the script is run in a newly opened PowerShell command shell, there is no error message at all, as shown by executing the statements:

```
$Error[0]
```

or:

```
$Error
```

Figure 18-4 shows the appearance when the code is run. Notice that the statement block in the for loop never runs.

Figure 18-4

The "error" of course isn't an error at all! It's simply a use of syntax that works in several languages but doesn't work in the same way in Windows PowerShell:

```
for ($i = 5; $i > 0; $i--)
```

There is no > operator (meaning greater than) in Windows PowerShell. In Windows PowerShell, the > operator means redirection. This is confirmed if you run the following statements:

```
get-childitem 0
get-content 0
```

As shown in Figure 18-5, a file named `0` has been created, and it contains the value of `5` (the value of the variable `$i` at the time of using the > redirection operator).

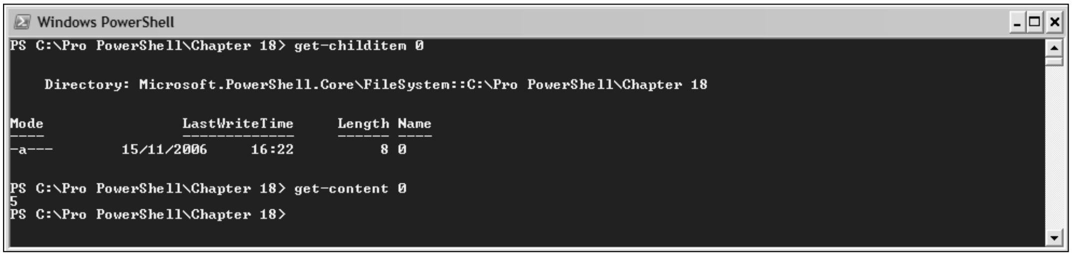


Figure 18-5

The `$i > 0` has been correctly interpreted by the Windows PowerShell parser as meaning that the value of the variable `$i` should be redirected to the file named `0` in the current location. And that is what Windows PowerShell did. If you change the offending line to

```
for ($i = 5; $i -gt 0; $i--)
```

and use the `-gt` comparison operator correctly, the script executes in the way intended.

The preceding behaviour is what happens with the Windows PowerShell debug facilities turned off. When the Windows PowerShell debugging functionality is turned on, the information displayed to you changes significantly.

When you use variable names, be careful that you don't include the `$` sign in the name. For example, in the statement

```
get-childitem Fred.txt -errorVariable $myErrorVar
```

you are not creating the variable `$myErrorVar`; in fact, you are not creating any error variable. If you make the preceding mistake instead of writing the following correct code:

```
get-childitem Fred.txt -errorVariable myErrorVar
```

then no variable `$myErrorVar` is created. If you then go on to use that nonexistent variable (that you assume exists) in some conditional logic, you are likely to get surprising and possibly frustrating results.

Some errors due to mistyping can lead to errors that are difficult to find. In isolation, you can probably spot fairly easily the error in the following statement:

```
$a = $ErrorActionPrefernce
```

I have omitted a single letter from `$ErrorActionPreference`, instead misspelling it as `$ErrorActionPrefernce`. There is no variable with the misspelled name, yet you see no error message when you execute the preceding statement, Even with `$ErrorActionPreference` (correctly spelled) set to `Stop`, no error is displayed and nothing is added to `$Error`.

Figure 18-6 shows the results of executing the preceding code.

Figure 18-6

The set-PSDebug Cmdlet

The `set-PSDebug` cmdlet turns Windows PowerShell script debugging on and off.

Members of the Windows PowerShell team have indicated a full script debugger is likely to be available in a future version of Windows PowerShell. As far as I am aware, no indication has been given of which version or of a likely timescale.

The `set-PSDebug` cmdlet supports the common parameters (described in Chapter 6) and the following parameters. Each of the listed parameters is optional. All are named parameters.

- `Trace` — Specifies how tracing is to be carried out. Permitted values for this property are the `Int32` values `0`, `1`, and `2`.
- `Step` — Step through code one statement at a time.
- Strict — Specifies that an exception should be thrown if a variable is referenced before it has been assigned a value.
- `Off` — Turn debugging off.

The values of the `Trace` parameter have the meanings shown in the following table.

Trace Parameter Value	Meaning
0	No debugging.
1	Trace script lines.
2	Trace script lines, variable assignments, function calls, and scripts.

The following commands show what debug information is (or is not) displayed when the simple assignment statement

```
$a = 10
```

is executed multiple times. First the statement is run with debug set to off; then, in succession, debug is set to `Trace 0`, `Trace 1`, `Trace 2`, and, finally, `Off` again.

As you can see in Figure 18-7, `Trace 0` is the same as setting debugging to `Off`. No debugging information is displayed. When debugging is set to `Trace 1`, the Windows PowerShell statement is echoed to the console. When debugging is set to `Trace 2`, variable assignments, function calls, and the execution of scripts are also displayed.

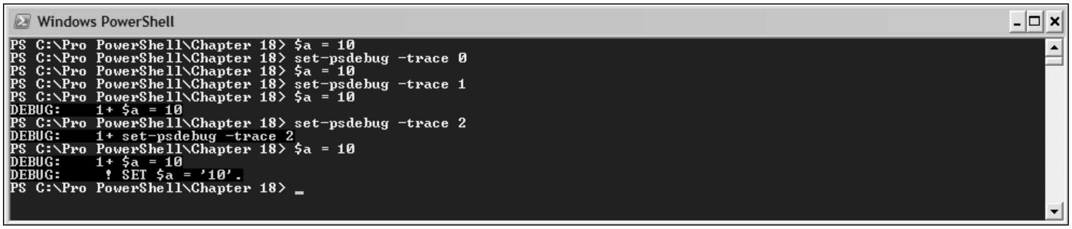


Figure 18-7

The script `Addition.ps1` contains a simple addition operation:

```
$a = 10
$b = 7
write-host "The sum of $a + $b is $($a+$b)"
```

With debugging set to `Trace 0`, the statements are simply echoed to the console, except the `write-host` statement, which requires the calculation of the sum of `$a` and `$b`, as indicated by `$($a + $b)`. This is shown as a separate step with debugging set to either `Trace 1` or `Trace 2`, as you can see in Figure 18-8. Similarly, the call to the script `Additions.ps1` is shown separately. With debugging set to `Trace 2`, each assignment statement is identified by the `SET` label and each call to a script is identified by a `CALL` label.

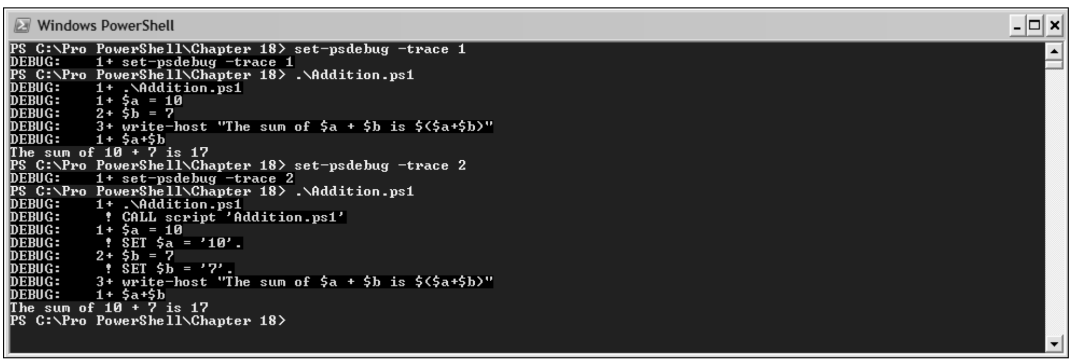


Figure 18-8

Once you have a mixture of functions and scripts, setting debugging to `Trace 2` can be helpful for following the flow of control. The following function, `FourLines`, writes two lines to the console and calls a simple function, `SayHello`, and then calls a script, `Hello.ps1`:

```
function FourLines {
write-host "First line in the function"
write-host "Second line in the function"
SayHello
.\Hello.ps1
}
```

Figure 18-9 shows the results of calling the `FourLines` function. Notice the three `CALL` labels in the debug material — first to the `FourLines` function itself, next to the `SayHello` function, then to the `Hello.ps1` script.

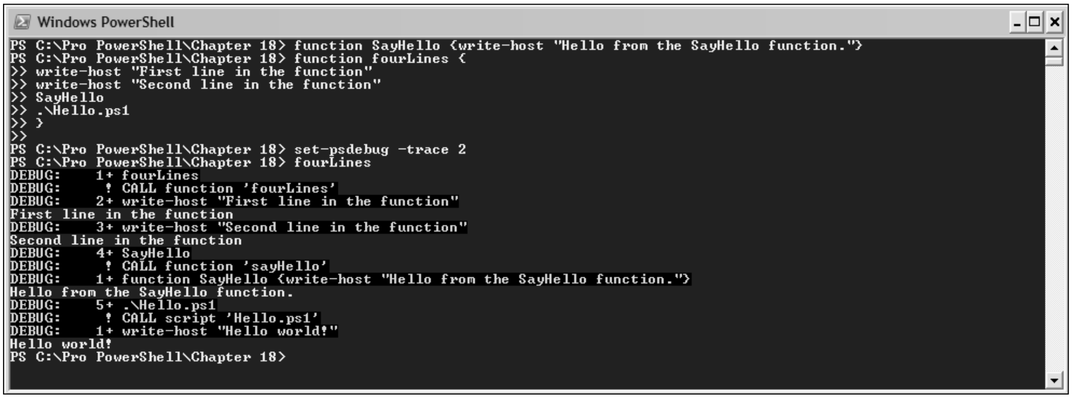


Figure 18-9

Setting `Trace` to `2` also specifically tracks `Trap` statements. Figure 18-10 shows the script `ForLoopTrapped.ps1` (which you saw in Chapter 17) run with `Trace` set to `2`. Notice that when `$i=0`, the error generated by executing `1/$i` is trapped. The trap statement is echoed in the debug output.

Notice that in the information about the `TRAP` statement, you are shown information about the error:

```
$? =True
DEBUG:     ! SET $i = '0'.
DEBUG:    4+ write-host '$i'"=$i"
$i =0
DEBUG:    5+ 1/$i
DEBUG:     ! TRAP generic; caught [System.DivideByZeroException]
DEBUG:    3+ trap {write-host "This is a custom error message.";continue}
This is a custom error message.
DEBUG:    3+ trap {write-host "This is a custom error message.";continue}
```

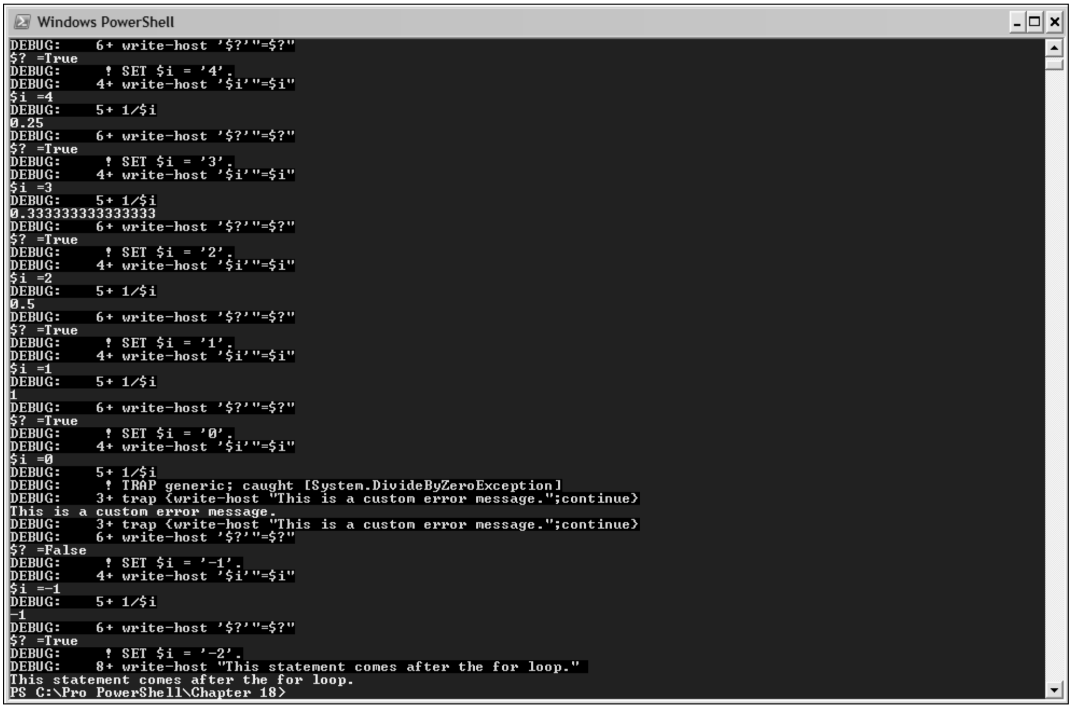


Figure 18-10

> *When you use the* `-step` *option, use the statement* `set-psdebug -off` *before you attempt a statement like* `clear-host`*. Alternatively, choose A, which means yes to all; otherwise, clearing the screen can be a slow task.*

When debugging is set to `Step` for `Addition.ps1`, the onscreen appearance and interaction are very different from any behavior set using the `-trace` parameter. After each statement, the user is asked if he wants the execution of a step (or all remaining steps) to continue or not and is also given an option to suspend the current shell. If the user chooses Yes, the line is executed and the debug information appropriate to the value of the `-trace` parameter is displayed. Figure 18-11 shows the appearance when answering Y (Yes) to each question posed step by step.

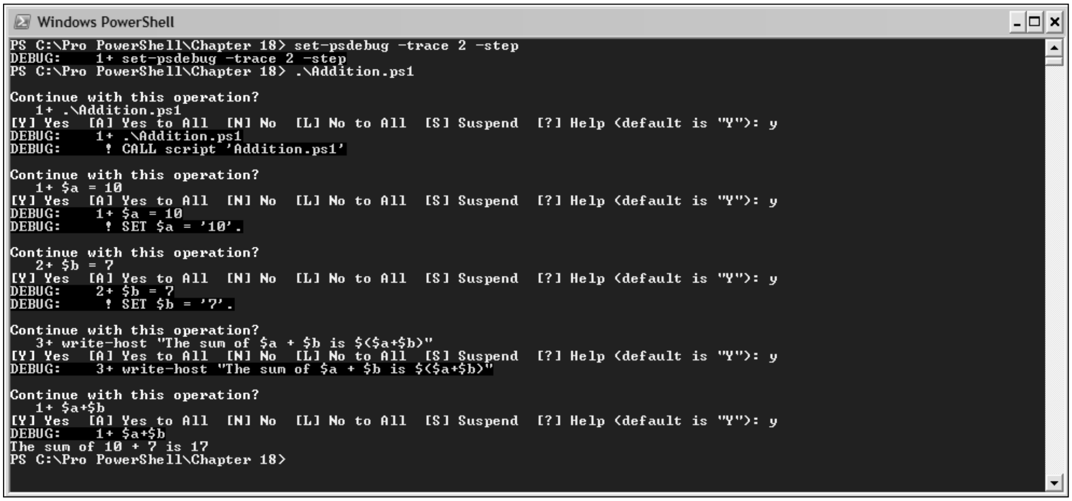


Figure 18-11

It's reasonably interesting to step through a very short script like `Addition.ps1`, but it becomes more interesting if you select the `Suspend` option at appropriate times.

After the statement:

```
$a = 10
```

I chose the `Suspend` option. Notice in Figure 18-12 that after selecting `Suspend` the prompt changes to include three successive chevrons (>>>). That indicates that a new PowerShell shell is being used.

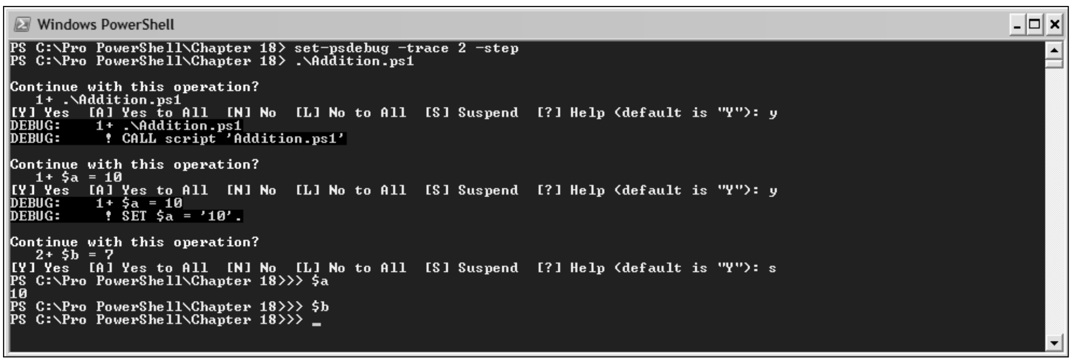


Figure 18-12

You can explore any relevant piece of information to help you understand what state, for example, variables are in. In this simple case, a value has been assigned to `$a` but not yet to `$b`. So, in the subshell you can type the statements

```
$a
$b
```

and the value of `$a` is, as expected, echoed to the screen. It has the value `10`, which you would expect at this stage of execution of the script. There is no value assigned yet to `$b`, so there is no value to echo to the screen.

Type `Exit` to return to the original shell. You are again asked if you want to execute the statement:

```
$b = 7
```

Select `[Y] Yes`. Then select `[S] Suspend` again. Now you can display the value for both `$a` and `$b`, as shown in Figure 18-13.

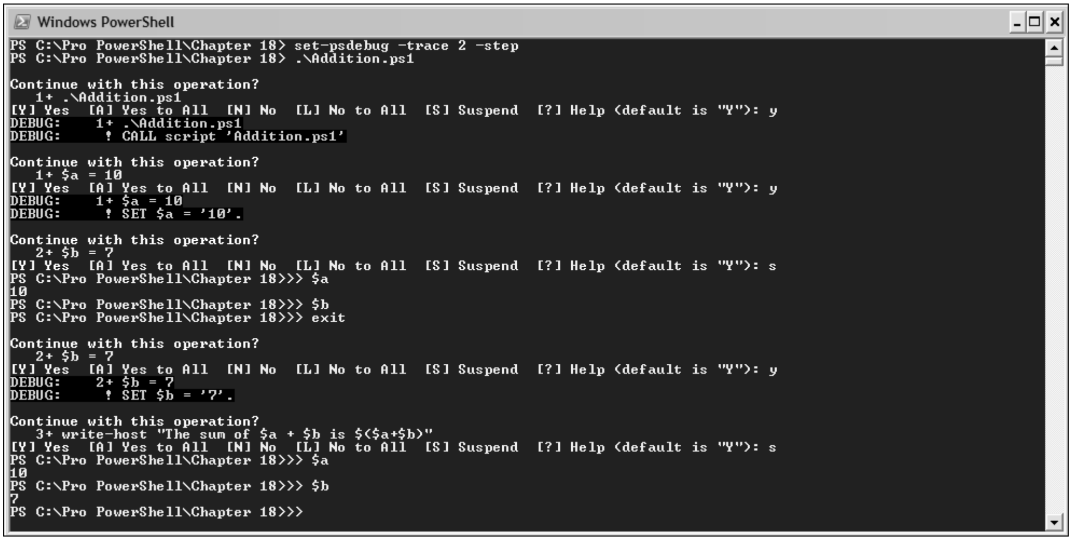


Figure 18-13

You might then exit again from the subshell and select `[A] Yes to All` to allow the script to finish executing.

Even for debugging a simple script such as `Addition.ps1`, stepping through a script one step at a time can be a tedious process. However, the capability to suspend execution of a script, enter a subshell, and inspect variables is a very powerful tool to debug Windows PowerShell scripts. Of course, it isn't as slick as the support for debugging in Visual Studio 2005, but it's a useful tool nonetheless.

The write-debug Cmdlet

The `write-debug` cmdlet writes a message to the Windows PowerShell console. It differs from the `write-host` cmdlet in that the effect of the `write-debug` cmdlet is controlled by the value of the

`$DebugPreference` variable. In addition to the common parameters, the `write-debug` cmdlet supports a single parameter:

- `Message` — The debug message to be sent to the console

The `Message` parameter is a required parameter and is also a positional parameter at position 1.

The file `AdditionWithDebug.ps1` has several `write-debug` statements that make it clear to the user which part of the script has been entered:

```
write-debug "Script addition.ps1 has been entered."
$a = 10
Write-debug "Variable $a has been assigned"
$b = 7
Write-debug "Variable $a has been assigned"
write-host "The sum of $a + $b is $($a+$b)"
write-debug "Execution of script addition.ps1 has been completed."
```

The default setting of the `$DebugPreference` variable is `SilentlyContinue`. If you want to see the results of `write-debug` statements displayed, set the value of the `$DebugPreference` variable to `Continue`.

```
$DebugPreference = "Continue"
```

With the value of `$DebugPreference` set to `Continue`, Figure 18-14 shows the results of executing the script `AdditionWithDebug.ps1`. The `set-psdebug` setting was `Off`.

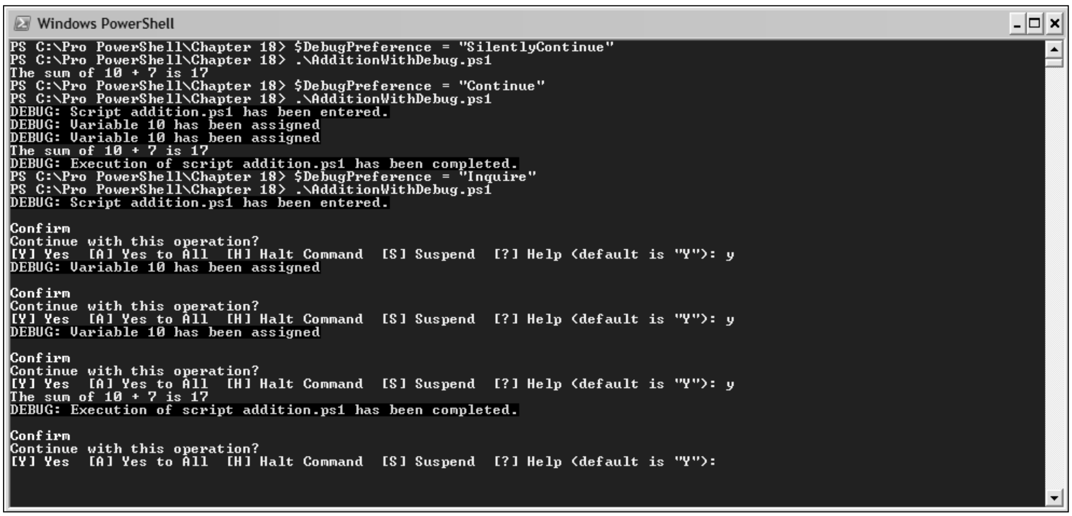


Figure 18-14

If you set `$DebugPreference` to `Stop`, then execution of a script will stop after the first `write-debug` statement, as shown in Figure 18-15.

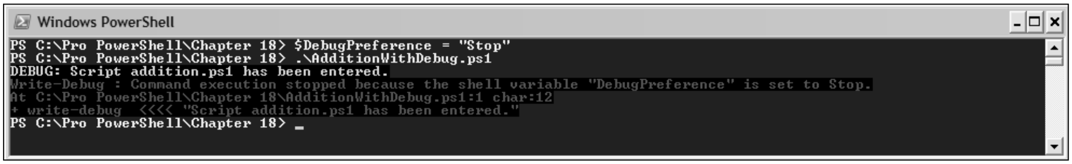


Figure 18-15

I find the Inquire setting of `$DebugPreference` very useful when I am actively debugging. It is more flexible than using

```
set-psdebug -step
```

since you can insert, delete, or comment out `write-debug` statements so that you can step through parts of a script that are of particular interest to you.

You can combine explicit `write-debug` statements by using the `set-psdebug` cmdlet. The information from `set-psdebug` and from explicit `write-debug` statements are interleaved as the script executes. Figure 18-16 shows the combining of output set by `set-psdebug` and `write-debug`.

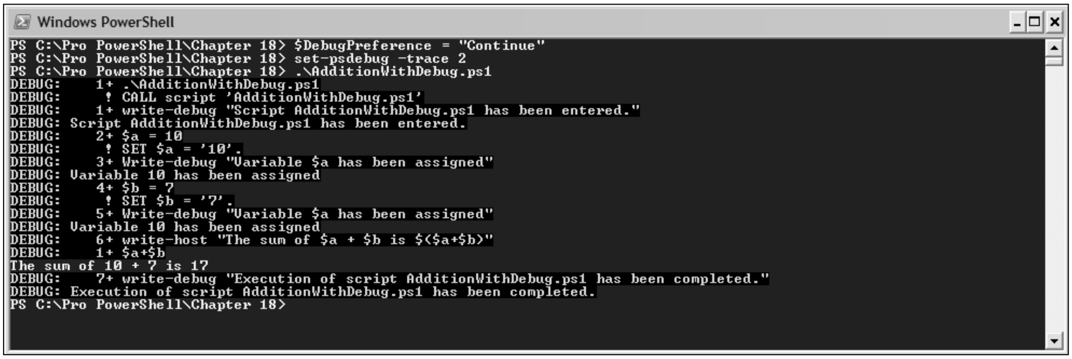


Figure 18-16

Combing output from `set-psdebug` and `write-debug` can be a bit overwhelming, but if you craft the `write-debug` statements appropriately, you can potentially avoid the need to run step by step through a script and simply display variables at times of interest.

If you compare the output on the line from `set-psdebug` that refers to the `write-debug` statement with the line that follows (that is produced by `write-debug`), then you can get a handle on the values of variables without adding explicit code to do that or exiting into a subshell to inspect the values of variables. The first of the following two lines is produced by `set-psdebug`. Because it outputs the code literally, you can see which variable is being referred to. In the following line, `write-debug` has executed and has output the value of the variable `$a`. Putting the two pieces of information together, you can see that `$a` has been assigned the value `10`.

```
DEBUG:    5+ Write-debug "Variable $a has been assigned"
DEBUG: Variable 10 has been assigned
```

You may prefer to make the debug information more explicit. The script `AdditionWithVariablesDebug.ps1` illustrates the kind of thing you can do using `write-debug` to display variable values.

```
write-debug "Script AdditionWithVariablesDebug.ps1 has been entered."
$a = 10
Write-debug "Variable $a has been assigned"
write-debug "The value of variable a is $a"
$b = 7
Write-debug "Variable $a has been assigned"
write-debug "The value of variable b is $b"
write-host "The sum of $a + $b is $($a+$b)"
write-debug "Execution of script AdditionWithVariablesDebug.ps1 has been
completed."
```

In `AdditionWithVariablesDebug.ps1`, I have included two statements that specify the value of the variables:

```
write-debug "The value of variable a is $a"
```

and:

```
write-debug "The value of variable b is $b"
```

As far as I can ascertain, you cannot use paired apostrophes (or paired escaped apostrophes) to display the name of a variable as `$a`. However, the two preceding statements allow you to verify the value of the variables `$a` and `$b` at specified points during script execution.

In the following excerpt from the output shown in Figure 18-17, the lines highlighted in gray are produced by `write-debug` statements. The lines with a white background are produced by `set-psdebug`.

```
DEBUG:    2+ $a = 10
DEBUG:     ! SET $a = '10'.
DEBUG:    3+ Write-debug "Variable $a has been assigned"
DEBUG: Variable 10 has been assigned
DEBUG:    4+ write-debug "The value of variable a is $a"
DEBUG: The value of variable a is 10
```

Figure 18-17 shows the result of executing the script `AdditionWithVariablesDebug.ps1`.

In a real-life situation you would be debugging much longer scripts than those I have used to illustrate the `write-debug` cmdlet. By using the `write-debug` statement, you can check the values of multiple variables or other values at critical points in script execution. This can be much quicker than stepping one step at a time through a long, complex script.

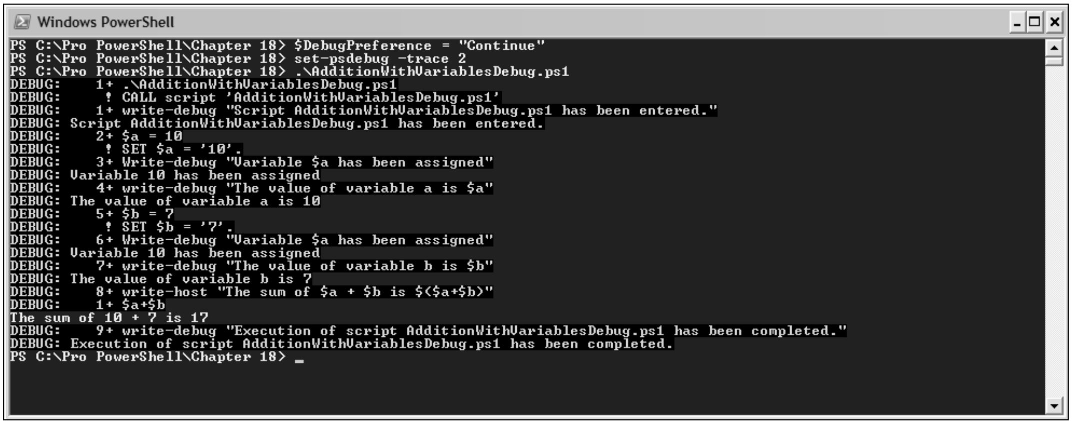


Figure 18-17

You can also embed `set-psdebug` statements in a script. The following script, `TraceAdditionWithVariablesDebug.ps1`, focuses the debugging attention around the assignment of a value to `$b`.

```
write-debug "Script addition.ps1 has been entered."
$a = 10
Write-debug "Variable $a has been assigned"
write-debug "The value of variable a is $a"
set-psdebug -Trace 2
$b = 7
Write-debug "Variable b has been assigned"
set-psdebug -Off
write-debug "The value of variable b is $b"
write-host "The sum of $a + $b is $($a+$b)"
write-debug "Execution of script addition.ps1 has been completed."
```

It uses the command

```
set-psdebug -Trace 2
```

to switch tracing on, and

```
set-psdebug -Off
```

to switch it off.

Figure 18-18 shows the results of executing `TraceAdditionWithVariablesDebug.ps1`.

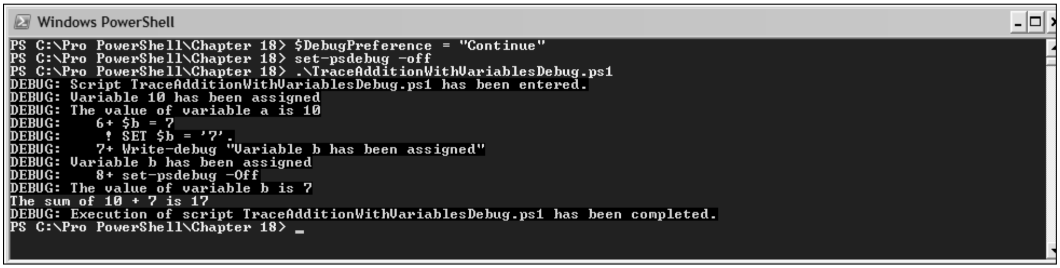


Figure 18-18

In the following excerpt from the output shown in Figure 18-18, the lines highlighted in gray are produced by `write-debug` statements. The lines with a white background are produced by `set-psdebug`, except for the second-to-last line, which is produced by a `write-host` statement.

```
DEBUG: Script TraceAdditionWithVariablesDebug.ps1 has been entered.
DEBUG: Variable 10 has been assigned
DEBUG: The value of variable a is 10
DEBUG:    6+ $b = 7
DEBUG:     ! SET $b = '7'.
DEBUG:    7+ Write-debug "Variable b has been assigned"
DEBUG: Variable b has been assigned
DEBUG:    8+ set-psdebug -Off
DEBUG: The value of variable b is 7
The sum of 10 + 7 is 17
DEBUG: Execution of script addition.ps1 has been completed.
```

Thus, by combining use of the `set-psdebug` and `write-debug` cmdlets, you can control what you display in selected parts of extensive scripts. By modifying the value of `$DebugPreference`, you can leave `write-debug` statements in your scripts but suppress the display of their content, until you are convinced that all the problems with a script have been ironed out. Simply set `$DebugPreference` to `SilentlyContinue` to conceal all `write-debug` statements from the user.

Tracing

Tracing functionality is provided with Windows PowerShell, but it is intended primarily for use by Microsoft support staff. However, if you want to explore beneath the covers what Windows PowerShell is doing, the tracing cmdlets can be very useful. However, the volume of information produced can quickly become daunting.

There are three such cmdlets:

- `trace-command`
- `set-tracesource`
- `get-tracesource`

I describe each of these cmdlets briefly in the following sections.

The trace-command Cmdlet

The `trace-command` cmdlet enables tracing of a specified trace source during the execution of a command.

In addition to supporting the common parameters, the `trace-command` cmdlet also supports the following parameters:

- `Name` — Specifies the `TraceSource` categories that tracing is to take place on. The parameter is required, and it is a positional parameter in position 1.
- `Expression` — Specifies the script code for which tracing will be carried out. The parameter is required and is a positional parameter in position 2.
- `Option` — Specifies the flags to be set on the `TraceSource`. An optional parameter that is a positional parameter at position 3. The default value of this parameter is `All`.
- `FilePath` — Adds the file trace listener using the specified file.
- `Debugger` — Adds the debugger trace listener if this parameter is specified.
- `PSHost` — Add the PowerShell Host trace listener if this parameter is specified.
- `ListenerOption` — Specifies the options for output from the trace listeners. The default value is `None`.
- `InputObject` — Specifies the current pipeline object to be handled when executing the expression.
- Force — If present, overrides normal restrictions.
- `Command` — Specifies the command for which tracing will be done.
- `ArgumentList` — Allows arguments to be set.

The volume of trace information can easily become daunting when using the `trace-command` cmdlet. To avoid long-running or nonterminating commands, use the `get-tracesource` cmdlet (described later in this chapter) to find the appropriate value for the `Name` parameter. Avoid using the `*` wildcard as the value of the `Name` parameter.

The allowed values for the `Option` parameter are

- `None`
- `All`
- `Assert`
- `Constructor`
- `Data`
- `Delegates`
- `Dispose`
- `Error`

- `Errors`
- `Events`
- `Exception`
- `ExecutionFlow`
- `Finalizer`
- `Lock`
- `Method`
- `Property`
- `Scope`
- `Verbose`
- `Warning`
- `WriteLine`

The allowed values for the `ListenerOption` parameter are

- `None`
- `Callstack`
- `DateTime`
- `LogicalOperationStack`
- `ProcessId`
- `ThreadId`
- `Timestamp`

Frequently used values for the `Name` parameter are

- `CommandDiscovery` — Shows the command discovery algorithm running
- `FormatFileLoading` — Shows the format from a `format.ps1 xml` file
- `FormatViewBinding` — Shows how a view is formatted
- `MemberResolution` — Show how members of an object are chosen when running a Windows PowerShell command
- `ParameterBinding` — Shows how parameters are bound to cmdlets
- `PathResolution` — Shows how wildcards are interpreted when resolving paths
- `RunspaceInit` — Shows what is happening during runspace initialization
- `TypeConversion` — Shows how one object is converted to a different type

The following command traces the `TypeConversion` information for casting a string to a `System.DateTime` object:

```
trace-command -Name TypeConversion -Expression {[DateTime]"2006/12/31"} -Options
All -PSHost
```

The results of executing the preceding command are shown in Figure 18-19.

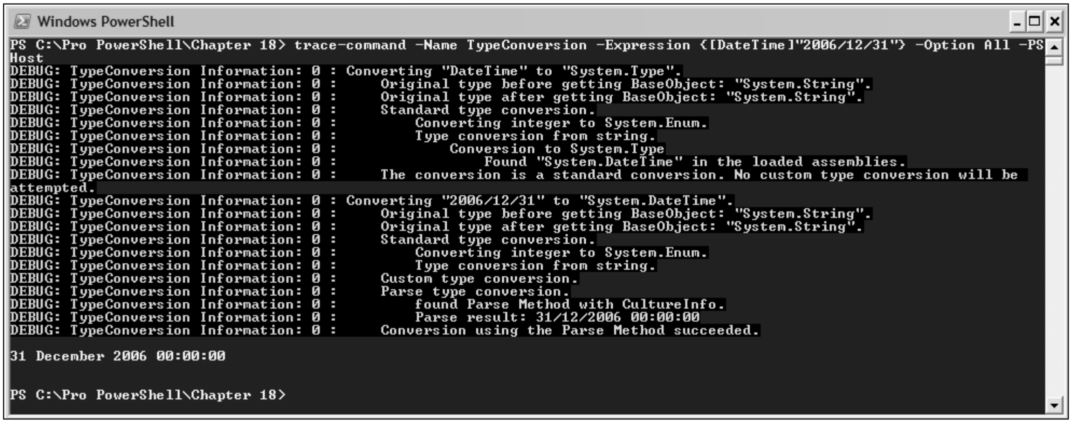


Figure 18-19

The following command traces metadata processing, parameter binding the creation and destruction of cmdlets relating to the running `svchost` processes:

```
trace-command -Name metadata, ParameterBinding, Cmdlet -Option All -Expression
{get-process svchost} -PSHost
```

Figure 18-20 shows the last of several screens of output.

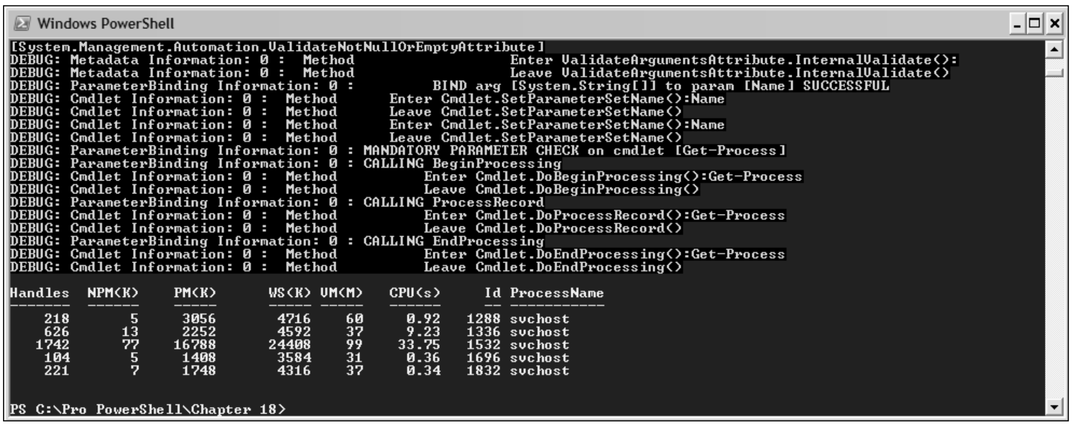


Figure 18-20

If you are interested in knowing more about what happens in Windows PowerShell under the covers, you could explore some more of the over 100 values allowed for the `Name` parameter (see the `get-tracesource` section, which follows later in this chapter).

The set-tracesource Cmdlet

The `set-tracesource` cmdlet sets or removes options or trace source listeners from a specified trace source instance.

In addition to the common parameters, the `set-tracesource` cmdlet supports the following parameters:

- `Name` — Specifies the trace source categories that will be affected. A required parameter that is also a positional parameter, taking position 1.
- `Option` — The flags to be set on the trace source. The allowed values are listed in the section on the `trace-expression` cmdlet.
- `FilePath` — Adds the file trace listener using a specified file.
- `Debugger` — Adds the debugger trace listener.
- `PSHost` — Adds the MSH host trace listener.
- `ListenerOption` — Specifies listener options.
- `PassThru` — Boolean. If true, the modified object is written to the pipeline.
- `RemoveListener` — Optional. If specified, removes all named listeners.
- `RemoveFileListener` — Optional. If specified, removes named file listeners.

The get-tracesource Cmdlet

The `get-tracesource` cmdlet lists properties for given trace sources.

In addition to the common parameters, the `get-tracesource` cmdlet supports one parameter:

- `Name` — Specifies trace sources

To find all the possible values for the `Name` parameter of the `trace-command` cmdlet, use this command:

```
get-tracesource -Name *
```

You can count the number of trace sources available to you using the following code:

```
$TraceSources = get-tracesource -Name *
$TraceSources.Count
```

In the version I am running at the time of writing, there are 173 values possible for the `Name` parameter.

The command

```
get-tracesource -Name Param*
```

returns information about all trace sources whose name begins with `Param`. The results of executing the preceding command are shown in Figure 18-21.

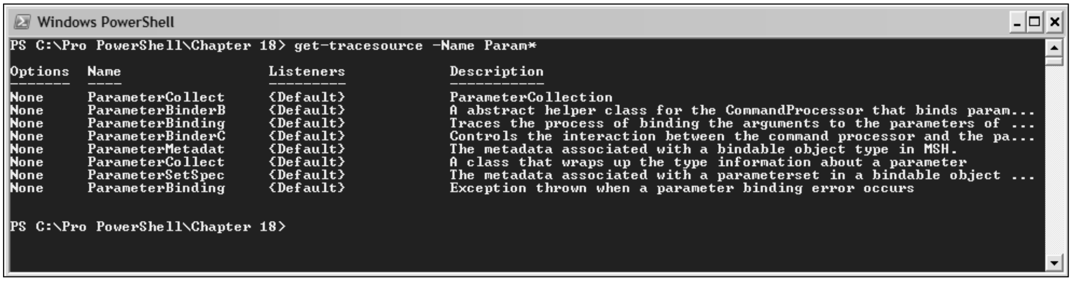


Figure 18-21

Summary

Debugging in Windows PowerShell is, like debugging in any other language, something of a black art. I discussed how simple syntax errors could cause potentially puzzling and sometimes silent errors.

This chapter introduced you to the `set-psdebug` cmdlet that allows you to set the volume of debugging information that is displayed. It also introduced the `write-debug` cmdlet, which allows you to output custom debug information, and showed you how to use the `$DebugPreference` variable to vary how `write-debug` statements are handled. In addition, it briefly described the `trace-command`, `set-tracesource`, and `get-tracesource` cmdlets.

19

Working with the File System

Windows PowerShell provides cmdlets to allow you to work effectively with drives, folders, and files on the file system. Windows PowerShell supports identification of drives using the `get-psdrive` cmdlet and exploration of files and folders using the `get-childitem` cmdlet. You can also create new drives, folders, and files. There is also a group of cmdlets that allow you to read and write content to and from text files.

Access to folders and files using Windows PowerShell is supported by the `FileSystem` provider. Additional command shell providers provide access to the HKLM (`HKey_Local_Machine`) and HKCU (`HKey_Current_User`) hives in the registry as well as drives for aliases, certificates, environment variables, functions, and variables. The command shell providers are in the `Microsoft.Management.Automation.Core` namespace.

If you want to find all providers supported on your system use the command:

```
get-psprovider
```

or, to find information about each drive and its associated provider, use this command:

```
get-PSdrive -Name * |
format-list
```

All drives on the local system are listed, as you can see in the part of the results shown in Figure 19-1.

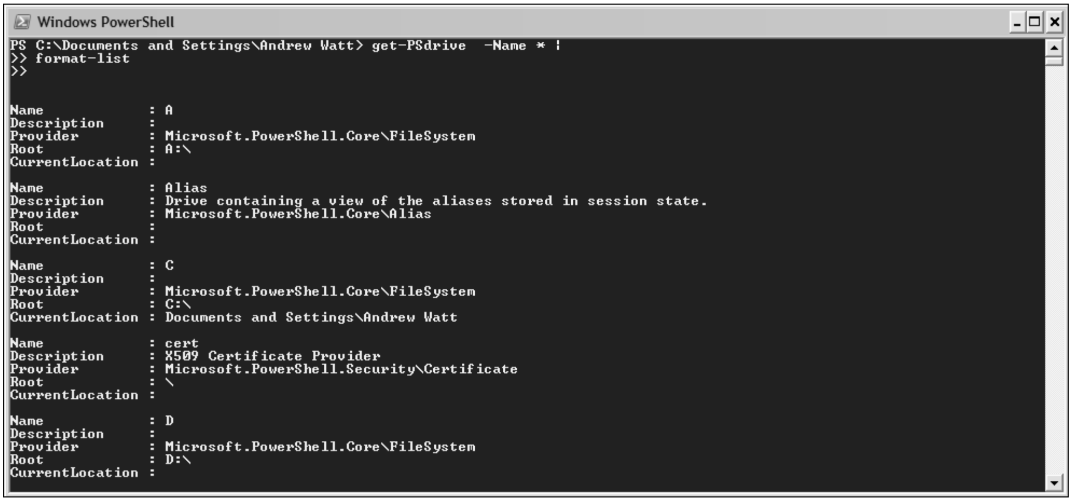


Figure 19-1

The default information displayed includes the command shell provider relating to each drive. The value of the `get-psdrive` cmdlet's `Name` parameter is the wildcard *, which matches all drives. The `format-list` cmdlet lets you see the full name of the provider that supports each drive.

To find all file system drives use this command:

```
get-psdrive -Name * -PSProvider FileSystem |
format-table Name, Root -auto
```

By specifying `FileSystem` as the value for the `-PSProvider` parameter, you indicate that only drives whose provider is the `FileSystem` provider are to be passed along the pipeline to the `format-table` cmdlet. There is no point in displaying the `Provider` column in the output (which would be displayed if you hadn't used the `format-table` cmdlet with specified columns) since, by specifying the value for the `-PSProvider` parameter in the second step of the pipeline, you know that the objects passed along the pipeline relate only to the `FileSystem` provider. The following shows the results on a system with one floppy drive, one hard drive, and one DVD drive.

```
Name Root
---- ----
A    A:\
C    C:\
D    D:\
```

Path Names in Windows PowerShell

When you specify a path name in Windows PowerShell you have two options — a fully qualified path name or a relative path name.

Fully Qualified Path Names

A fully qualified path name differs from absolute paths you may be familiar with in other contexts, since it may include the name of the Windows PowerShell provider that enables file system operations. A fully qualified name takes this form:

```
ProviderName::drive:\container\...\item
```

The preceding syntax applies to paths in all drives exposed by Windows PowerShell, not just those in the file system. In the context of the `FileSystem` provider the container is a folder (directory) and the item is a file.

An optional provider name is followed by a pair of colon characters if the provider name is used. Strictly speaking, the provider name is never needed, since drive names should be unique across a system. The drive name is followed by a single colon character and a backslash. Actually, Windows PowerShell will support the forward slash, too, if you are used to that convention due to a Unix or Linux background. Optionally, additional subcontainers (folders) can be included, as appropriate, in the path. The item is also optional, when the fully qualified path refers to a folder.

The following command lists all folders (and any files) in the `C:\Program Files` folder that begin with the character sequence `mi`:

```
get-childitem "FileSystem::C:\Program Files\mi*"
```

If you prefer, you can type the command using forward slashes and obtain the same result:

```
get-childitem "FileSystem::C:/Program Files/mi*"
```

The `item` (using the `FileSystem` provider an item refers to a file) is specified using a definition that includes a wildcard, `mi*`, which matches any name beginning with the character sequence `mi`. Since the item is optional folders in `C:\Program Files`, which begin with the character sequence `mi` are also displayed.

When you want to access a path that includes one or more space characters, you must enclose the path name in paired double quotation marks or paired apostrophes.

If you want to include folders in the results or expect folders in the results be careful how you use the `*.*` type of wildcard that you may have used often in `CMD.exe`. For example, compare in Figure 19-2 how, when using Windows PowerShell, the wildcard `mi*.*` returns many fewer folders than the earlier command, which used `mi*`.

```
get-childitem "FileSystem::C:\Program Files\mi*.*"
```

The difference in the number of returned folders isn't surprising, since `mi*.*` means find any name that begins with the character sequence `mi`, then has zero or more characters, then a literal period character, then zero or more characters. In the `Program Files` folder on my machine only one folder, `Microsoft.NET`, matches `mi*.*`, since it begins with the character sequence `mi` and also includes a literal period character. Figure 19-2 shows the results of running the two preceding commands.

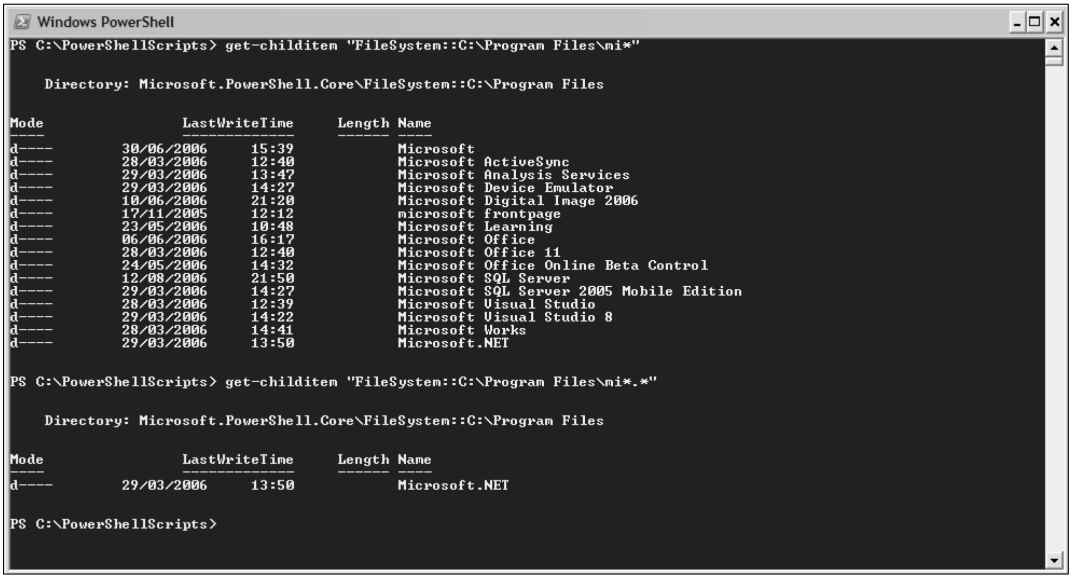


Figure 19-2

However, notice that Windows PowerShell differs from `CMD.exe` in how `mi*.*` is interpreted when looking for folders and files. As you can see in Figure 19-3, `CMD.exe` returns all folders beginning with the character sequence `mi` even when the pattern is `mi*.*`. My opinion is that the `CMD.exe` behavior is incorrect and that the Windows PowerShell behavior is correct. In any case, if you used wildcards such as `*.*` in `CMD.exe`, you need to be aware that PowerShell behavior is different.

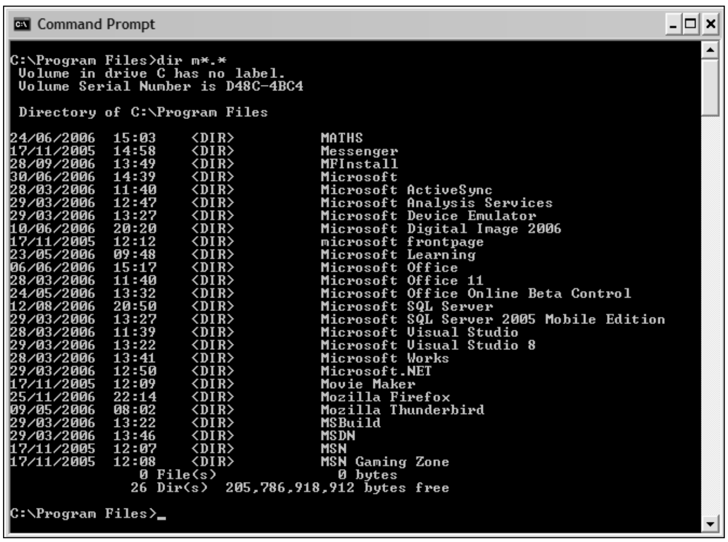


Figure 19-3

Windows PowerShell supports wildcards in parameter values beyond the traditional ? (matches any one character), and * (matches zero or more characters) metacharacters. You can specify a class of characters to match. A *character class* is signified by characters inside paired square brackets. For example,

```
get-childitem -Name C:\[abc]*
```

will match any folder or filename in the root of drive C: beginning with a or b or c. The characters inside the paired square brackets are matched once; then zero or more characters are matched.

When you use wildcards in the value of a parameter, they are case-insensitive.

The preceding command would match any of the following, if they were present in the current working directory:

```
apple.txt
bear.txt
cat.txt
```

since the first character of each filename is found in the character class [abc].

Windows PowerShell wildcards also support a *range* inside a character class. For example,

```
get-childitem -Name C:\[a-f]*
```

matches folders or filenames in the root of drive C: that begin with a through f.

The order of characters in a wildcard range is important. You must specify the character earlier in the alphabet first, then the hyphen, then the character last in the alphabet. Attempting to use [f-a] *as a range, for example, produces an error message.*

You can also use the hyphen as a literal character (rather than having a special meaning to signify a range) and use numeric digits inside a character class, too. To demonstrate this, you can create a few sample files using the following commands, which to send a short piece of text to a named. The following commands assume that the directory Pro PowerShell\Chapter 19 exists. Amend the path to suit, if you wish:

```
"test" > "\Pro PowerShell\Chapter 19\-hyphenfirst.txt"
"test" > "\Pro PowerShell\Chapter 19\1onefirst.txt"
"test" > "\Pro PowerShell\Chapter 19\2twofirst.txt"
"test" > "\Pro PowerShell\Chapter 19\Test1.txt"
"test" > "\Pro PowerShell\Chapter 19\Test2.txt"
"test" > "\Pro PowerShell\Chapter 19\Test3.txt"
```

To find files whose name begins with a hyphen or the numeric digit 2, use the following command:

```
get-childitem "\Pro PowerShell\Chapter 19\[-2]*.txt"
```

If you intend a hyphen inside the square brackets of a character class to match a hyphen (rather than to represent a range), then the hyphen must be the first character after the left square bracket.

Similarly, to match `Test1.txt` and `Test3.txt`, you can use a character class containing numeric digits as in the following command:

```
get-childitem "\Pro PowerShell\Chapter 19\Test[13].txt
```

Figure 19-4 shows the results of running the two preceding commands.

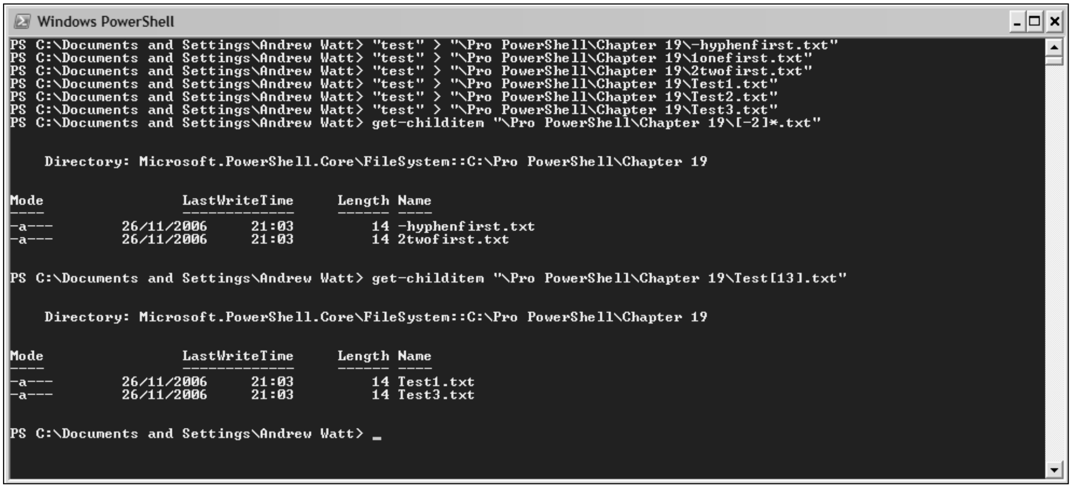


Figure 19-4

Relative Path Names

Relative path names are likely to be familiar to you. Given a specific current location a path is interpreted relative to that current location.

By default the PowerShell prompt displays the current directory. If you have configured the prompt so that the current location is not displayed as part of the prompt, you can display the current location using the command:

```
get-location
```

If the current location is `C:\` then, using the relative path, the command

```
get-childitem "Program Files\mi*.*"
```

finds all folders and files that match the item name (including wildcards). The equivalent command using an absolute path is:

```
get-childitem "C:\Program Files\mi*.*"
```

Figure 19-5 shows the use of the fully qualified path name and relative path name.

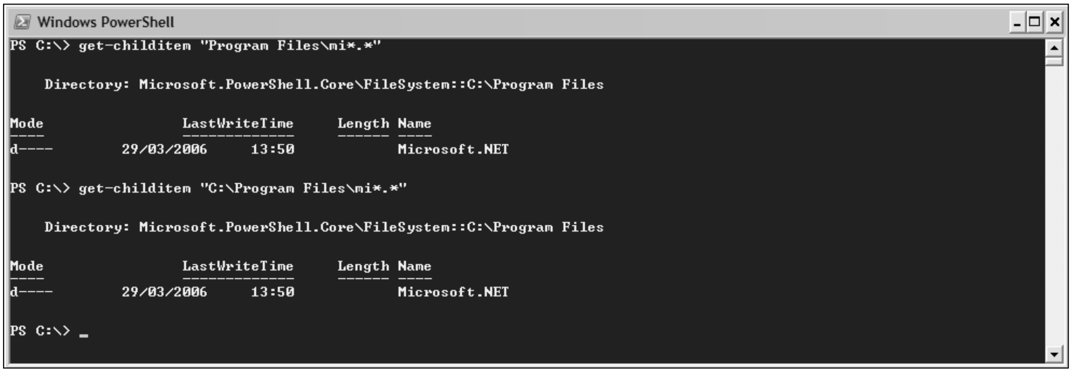


Figure 19-5

The Windows PowerShell notation for current location (a single period character) and the parent of the current location (two period characters) is likely to be familiar to you from `CMD.exe`.

Path Names and Running Commands

When you specify a fully qualified path name for a command or specify a relative path name for a command, Windows PowerShell searches for a matching command in different ways.

If you use a fully qualified path name, Windows PowerShell looks for a matching filename in the specified location. If such a file is found, the command is run (subject to security settings). If no matching file is found in the specified location Windows PowerShell runs no command.

When using a fully qualified path name without any spaces, you can simply type:

```
C:\SomeDirectory\SomeScript
```

or:

```
C:\SomeDirectory\SomeScript.ps1
```

to run a script called `SomeScript.ps1` located in the `SomeDirectory` folder (assuming that the necessary permissions are in place to run scripts on the machine).

If, however, the path contains one or more space characters, for example `C:\Pro PowerShell\Chapter 19\SomeScript.ps1`, you have to use paired quotation marks or apostrophes to avoid an error message about `C:\Pro` not being a recognized cmdlet and so on. However, when you type:

```
"C:\Pro PowerShell\Chapter 19\SomeScript.ps1"
```

or:

```
"C:\Pro PowerShell\Chapter 19\SomeScript"
```

all that happens is that the string you typed is echoed back to the console, which is not surprising since you simply entered a string enclosed in paired quotes or apostrophes and Windows PowerShell is treating it simply as a string. To run the script add an ampersand, `&`, at the beginning of the line. The & character indicates to the PowerShell parser that what follows is to be treated as a command. PowerShell will run the script whether you put a space character between the ampersand and the path:

```
& "C:\Pro PowerShell\Chapter 19\SomeScript.ps1"
```

or omit the space character:

```
&"C:\Pro PowerShell\Chapter 19\SomeScript.ps1"
```

You don't need to include the period character and the `ps1` file extension to run the script.

Figure 19-6 shows the results when `SomeScript.ps1` contains a single command, `get-date`.

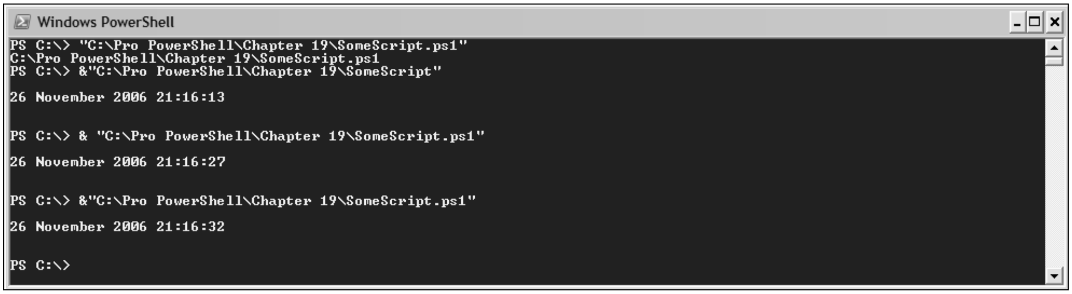


Figure 19-6

If script execution on the machine is restricted, you won't be able to run any scripts. You first need to alter Powershell's Execution policy, as described in Chapter 10.

When you use a command without specifying a fully qualified path name, Windows PowerShell searches the following locations for possible matches:

1. The aliases drive for currently defined aliases
2. The functions drive for currently defined functions
3. Commands in any folder specified in the `PATH` environment variable.

When you use a relative path, you need to be careful how you type the command if the script is in the current working directory. For example, typing:

```
SomeScript
```

or:

```
SomeScript.ps1
```

will produce the error message shown in Figure 19-7 (where the command is aimed at running the `SimpleScript.ps1` script.

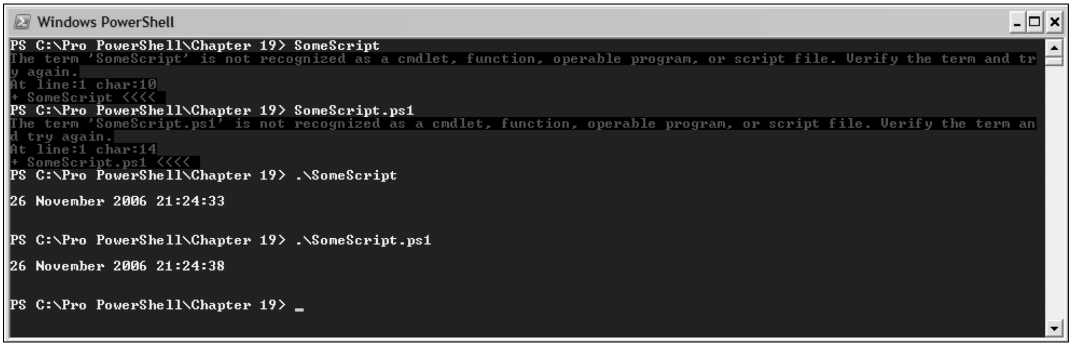


Figure 19-7

To successfully run a script in the current working directory, you need to explicitly specify that it is in the current directory, using the period notation to specify the current directory. So, to run a script named `SimpleScript.ps1` in the current directory use:

```
.\SimpleScript
```

or:

```
.\SimpleScript.ps1
```

The preceding behavior is intended as a security feature. If, for example, a virus or similar malware saved PowerShell scripts perhaps called something like `dir.ps1`, then typing `dir` at the command line wouldn't cause the malicious script to be executed. The protection is limited, since malware could modify an alias, as in the following sequence of commands:

```
remove-item alias:dir
 "write-host 'This script could have done something nasty.'" > C:\Nasty.ps1
 new-alias dir C:\Nasty.ps1
 dir
```

Figure 19-8 shows the results of running the preceding commands. When the user types `dir`, presumably to list files, the potentially malicious script is executed (subject to permissions to run scripts).

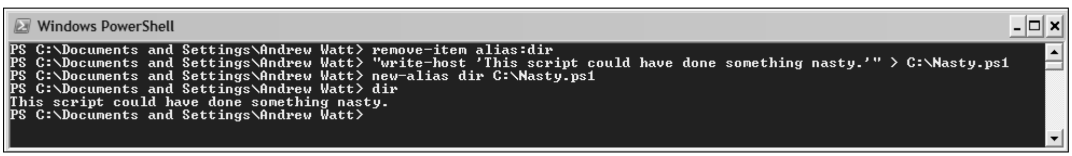


Figure 19-8

Simple Tasks with Folders and Files

This section demonstrates techniques you can use to explore drives, folders, and files on a Windows machine.

Finding the drives on a system

To find the drives on a machine, you use the `get-psdrive` cmdlet. Typically, simply typing

```
get-psdrive
```

displays all drives on the system, including exposing drives for the registry, environment variables, aliases, and functions. Figure 19-9 shows the drives on a Windows XP machine that has one hard drive.

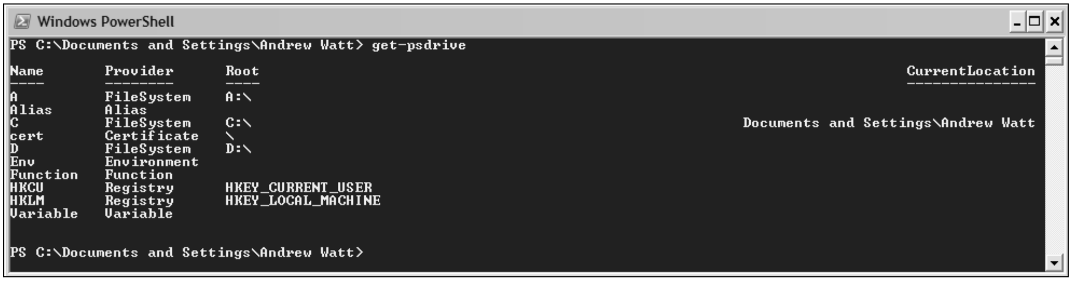


Figure 19-9

Finding Folders and Files

To find folders and files use the `get-childitem` cmdlet and specify the `Path` parameter to match a single folder or file or multiple folders or files. The `get-childitem` cmdlet returns `FileInfo` or `DirectoryInfo` objects when used with the `FileSystem` provider. When files are found `FileInfo` objects are returned. When folders (directories) are found, then `DirectoryInfo` objects are returned.

If you want to find only folders you can use the following command:

```
get-childitem * |
where-object {$_.Mode -match "d"}
```

Or to find files use this command:

```
get-childitem * |
where-object {$_.Mode -notmatch "d"}
```

The `where-object` cmdlet in the second step of the pipeline looks for a match in the `Mode` property of the `FileInfo` objects or `DirectoryInfo` objects passed on from the first pipeline step. Figure 19-10 shows the result of executing the preceding commands.

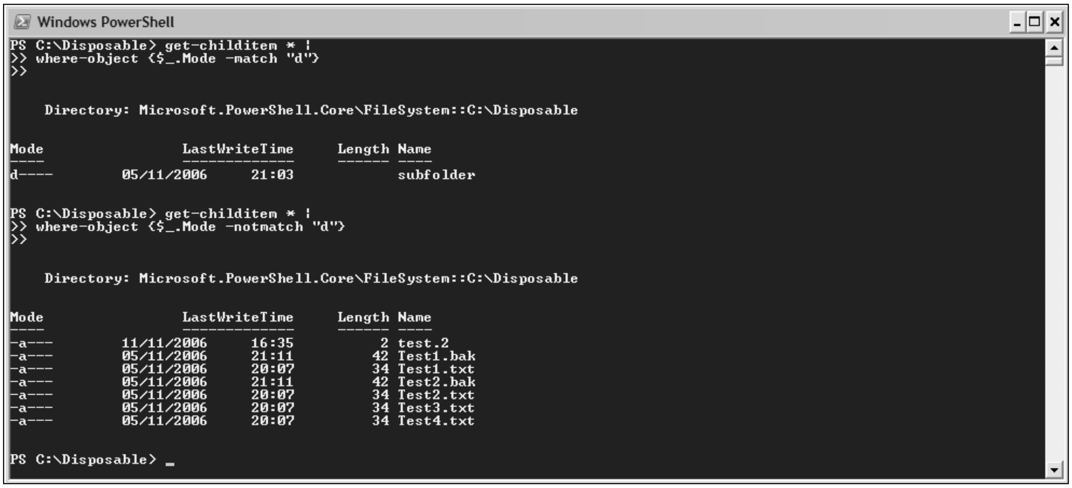


Figure 19-10

An alternative way to selectively retrieve directories is to use this command:

```
get-childitem * |
where-object {$_.GetType().Name -eq "DirectoryInfo"}
```

In the second step of the pipeline, the `GetType()` method of the current object (which is either a `DirectoryInfo` or a `FileInfo` object when the `FileSystem` provider is used) is used to retrieve the type of the object. Its `Name` property is then tested for equality to the string `DirectoryInfo` to test if the object is a `DirectoryInfo` object.

To selectively retrieve files use this command:

```
get-childitem * |
where-object {$_.GetType().Name -eq "FileInfo"}
```

This time the `Name` property is tested for equality to the string `FileInfo`. If that test returns `True`, then the object is a `FileInfo` object. Figure 19-11 shows the results returned from the preceding two commands.

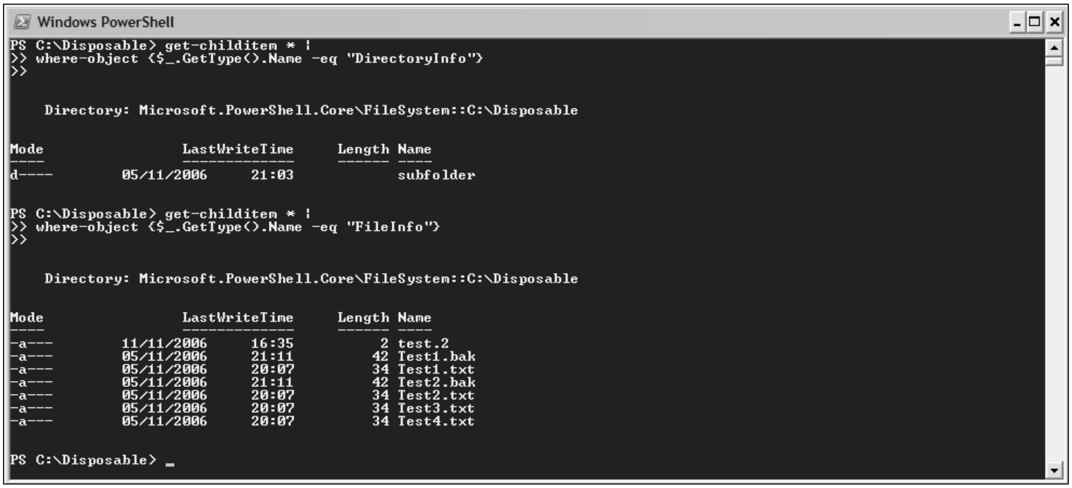


Figure 19-11

Finding File Characteristics

You might want to find information about the characteristics of a file that are not accessible simply by using the `get-childitem` cmdlet.

Working with FileInfo Object methods

To find the methods available on a `FileInfo` object, use this command:

```
get-childitem * |
where-object {$_.GetType().Name -eq "FileInfo"} |
get-member -memberType Method
```

There are 47 methods available on a `FileInfo` object. The `set_IsReadOnly()` method is used in the following example. In the following example, you will set a file, `ChangeAccess.txt` to read-only, attempt to append text to it (which fails), change it back to read-write, and then successfully write the appended text to the file.

First redirect some literal text to create a new file:

```
"Create as read-write" > ChangeAccess.txt
```

Then assign the `FileInfo` object for the file, retrieved by the `get-childitem` cmdlet to the variable `$a`:

```
$a = get-childitem ChangeAccess.txt
```

Then display the mode property of the file:

```
$a.Mode
```

Then set the file to read only using the `set_IsReadOnly()` method of the `FileInfo` object:

```
$a.set_IsReadOnly(1)
```

Then display the mode to confirm the new characteristic of the file:

```
$a.Mode
```

The `r` in the mode indicates that the file is now read-only.

Then attempt to redirect some text to the file using the >> redirection operator, which appends text to an existing file:

```
"Attempt to append text to the file" >> ChangeAccess.txt
```

The error message shown in Figure 19-12 is displayed. Then turn read-only off using the `set_IsReadOnly()` method of the `FileInfo` object with argument of 0:

```
$a.set_IsReadOnly(0)
```

And confirm the change using the `FileInfo` object's `Mode` property:

```
$a.Mode
```

The `r` is no longer present in the mode of the file, indicating that it is now read-write. Now retry to append text to the file:

```
"Attempt to append text to the file" >> ChangeAccess.txt
```

This time it succeeds, since the error message is no longer displayed. Confirm that the command has succeeded using the `get-content` cmdlet to echo the content of the file to the console:

```
get-content ChangeAccess.txt
```

Figure 19-12 shows the results at each step in the preceding example.

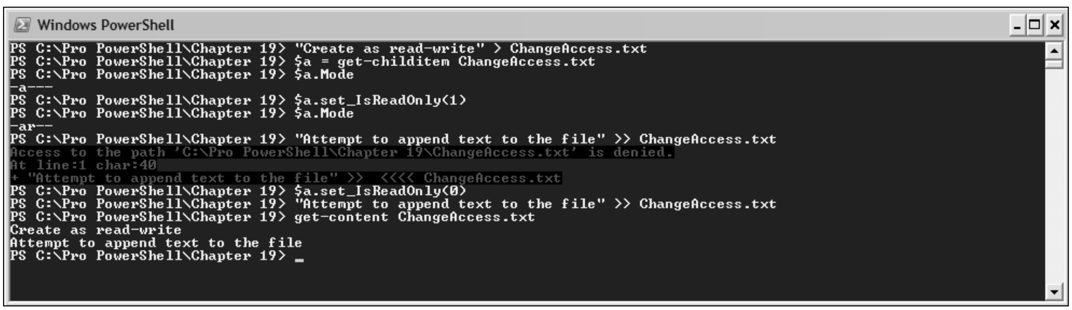


Figure 19-12

Working with FileInfo Object properties

To find the properties of a `FileInfo` object, use the following command:

```
get-childitem * |
where-object {$_.GetType().Name -eq "FileInfo"} |
get-member -memberType Property
```

The following example allows you to use some of the properties of the `FileInfo` object. In preparation for finding the files, first create three `.txt` files using the `add-content` cmdlet. The commands assume that `C:\Pro psh\Chapter 19` is the current directory. The script is called `CreateTextFiles.ps1`.

```
add-content Test1.txt "This is test file 1."
add-content Test2.txt "This is test file 2, you know."
add-content Test3.txt "Believe it or not this is test file 3."
```

Execute the `CreateTextFiles.ps1` script using this command:

```
.\CreateTextFiles.ps1
```

The following code uses the `foreach` statement to display selected characteristics of a collection of files:

```
foreach ($file in (get-childitem -Path "c:\Pro PowerShell\Chapter 19\Test*.txt"))
        {
            write-host $file.fullname " .... File length is: $($file.length)."
        }
```

Figure 19-13 shows the result of executing the preceding commands.

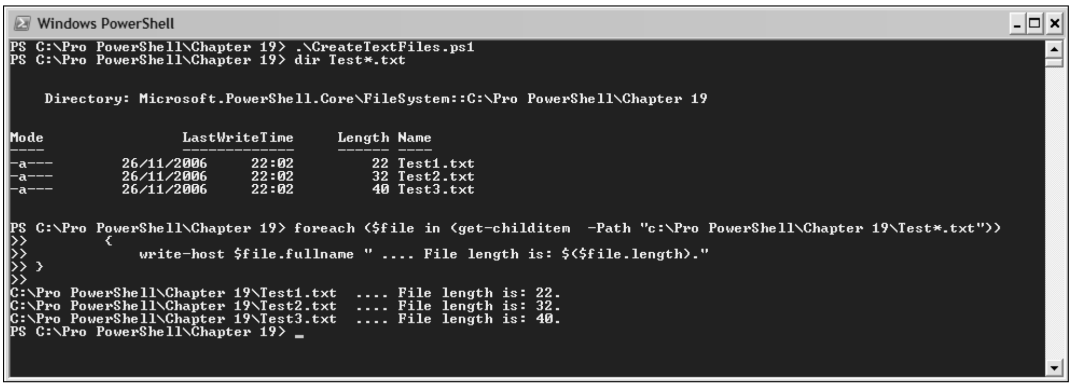


Figure 19-13

You can also use the `measure-object` cmdlet to display statistics about text files. For example, the following command displays the sum, average, minimum, and maximum lengths of the three files created using the `crateTextFiles.ps1` script.

```
get-childitem Test*.txt |
measure-object -Property Length -Average -Sum -Min -Max
```

Objects corresponding to the three text files are passed to the second step in the pipeline. The `Property` parameter of the `measure-object` cmdlet specifies which characteristic of the `FileInfo` objects are to be measured. The count is displayed by default. The `Average`, `Sum`, `Min`, and `Max` parameters cause the average, sum, minimum, and maximum values of the selected property to be displayed. Figure 19-14 shows the result of running the preceding command.

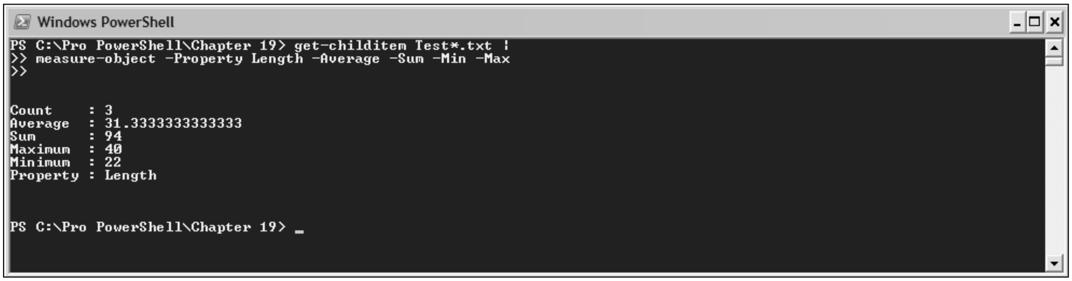


Figure 19-14

Exploring Files Using the select-object Cmdlet

The `select-object` cmdlet allows you to explore files and display ranked files according to some specified criterion. For example, the following simple pipeline using the `select-object` cmdlet

```
get-childitem "C:\Windows\System32\*.dll" |
select-object -First 5
```

returns the first five DLLs in the `Windows\System32` directory. By default, you see the first five files alphabetically. Similarly, using the command:

```
get-childitem "C:\Windows\System32\*.dll" |
select-object -Last 5
```

you see the last five DLL files ordered alphabetically. Figure 19-15 shows the results of running the two preceding commands on a Windows XP SP2 machine.

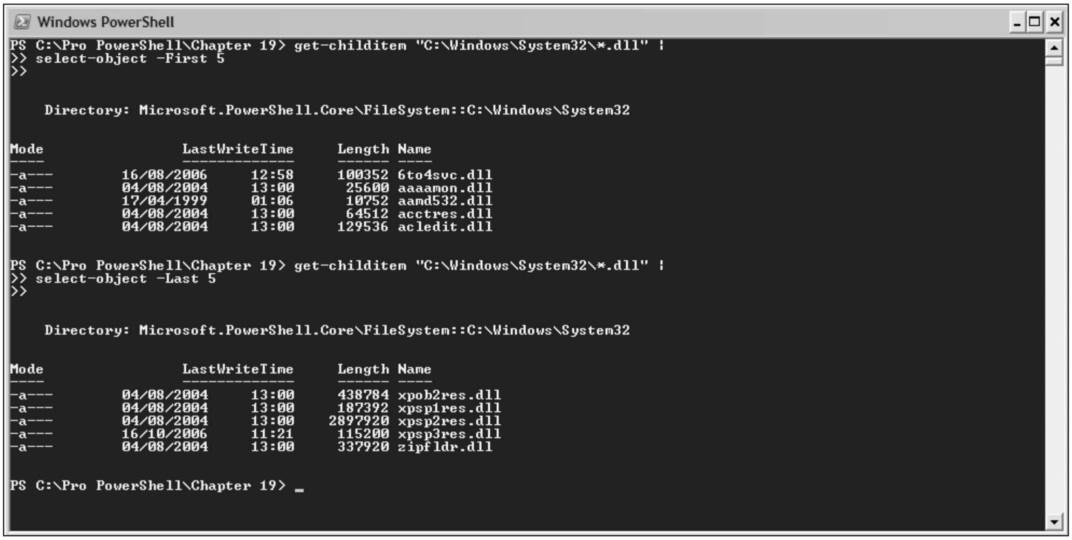


Figure 19-15

Of course, you don't need to choose five objects; you can specify whatever number is relevant to your needs. I chose that number to make it easy to display the results of the preceding commands on the page. Nor do you have to select the first and last files according to alphabetical order. You could specify any criterion that is of interest to you. For example, you could select files according to the file size, which is represented by the `Length` property on the `FileInfo` object. To find the first five files by size (that is the five smallest DLLs), use this command:

```
get-childitem "C:\Windows\System32\*.dll" |
sort-object Length |
select-object -First 5 |
format-table FullName, Length -auto
```

The first step in the pipeline selects all DLLs in the `Windows\System32` directory. The second step uses the `sort-object` cmdlet to sort the pipeline objects according to the value of the `Length` property. The third step uses the `select-object` cmdlet to select the first five objects passed from the second step. Since the second step has sorted the objects according to their size, the third step selects the five smallest files. The fourth step uses the `format-table` cmdlet to display the `FullName` and `Length` properties of the `FileInfo` objects.

To select the five largest DLLs in the directory, use the `Last` parameter with the `select-object` cmdlet:

```
get-childitem "C:\Windows\System32\*.dll" |
sort-object Length |
select-object -Last 5 |
format-table FullName, Length -auto
```

Figure 19-16 shows the results of running the two preceding commands.

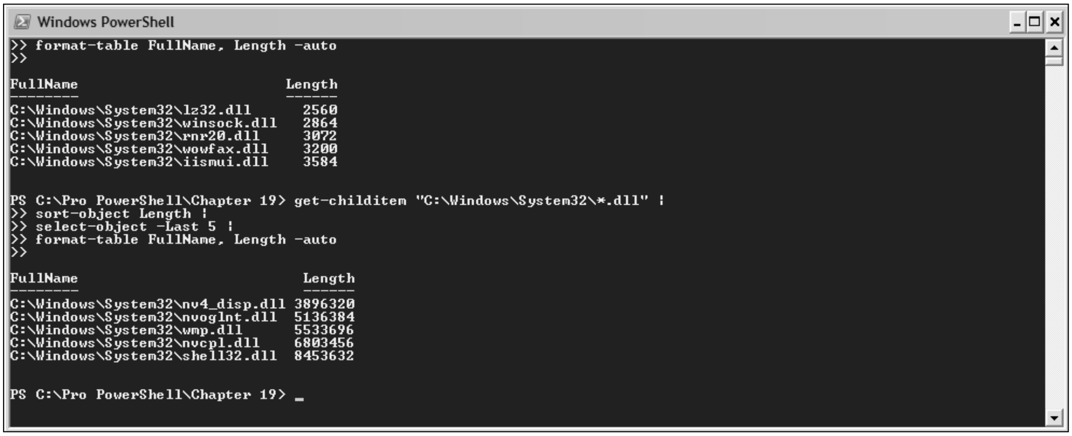


Figure 19-16

By modifying the property on which the `sort-object` cmdlet sorts in the second step of the pipeline, you can use the `select-object` cmdlet to display a sorted list based on any property of a `FileInfo` object. For example, if you want to find which files have not been accessed for longest use this command:

```
get-childitem "C:\Windows\System32\*.dll" |
sort-object LastAccessTime |
select-object -First 5 |
format-table FullName, LastAccessTime -auto
```

Or to find the most recently accessed files, use this command:

```
get-childitem "C:\Windows\System32\*.dll" |
sort-object LastAccessTime |
select-object -Last 5 |
format-table FullName, LastAccessTime -auto
```

In the second step of the pipeline, the objects are sorted on the value of the `LastAccessTime` property of the `FileInfo` object. Figure 19-17 shows the results of running the two preceding commands.

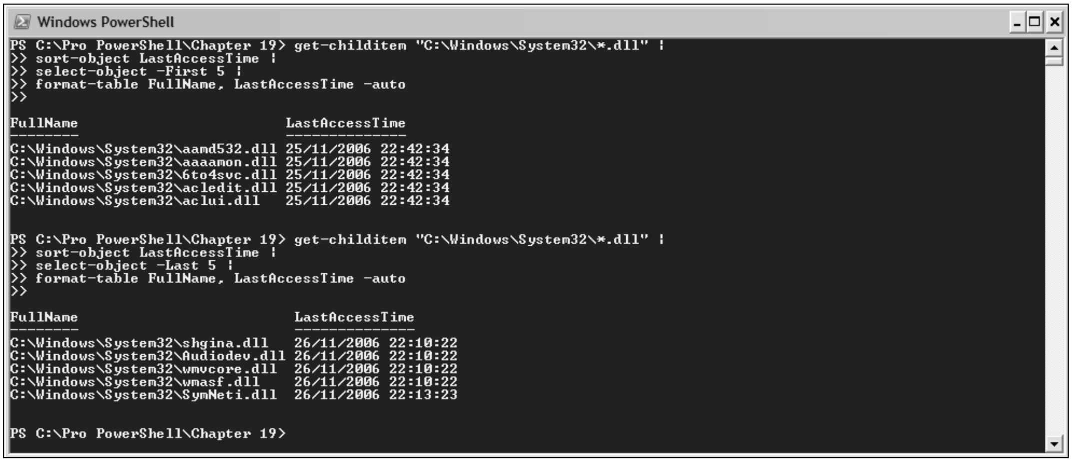


Figure 19-17

Finding Hidden Files

The `get-childitem` cmdlet allows you to force information about hidden files or folders to be displayed. To display hidden files or folders, use the `Force` parameter with the `get-childitem` cmdlet.

In this example, you display files and folders whose name begins with the letter `r`. The current location is `C:\` on a Windows XP SP2 machine. To display those files and folders use this command:

```
get-childitem -Path [rs]*
```

When you add the `Force` parameter:

```
get-childitem -Path [rs]* -Force
```

the hidden folders RECYCLER and System Volume Information are displayed in the results. Figure 19-18 shows the results of executing the two preceding commands. Notice in Figure 19-18 that the mode for RECYCLER and System Volume Information contains an `h` that indicates that the folder is hidden.

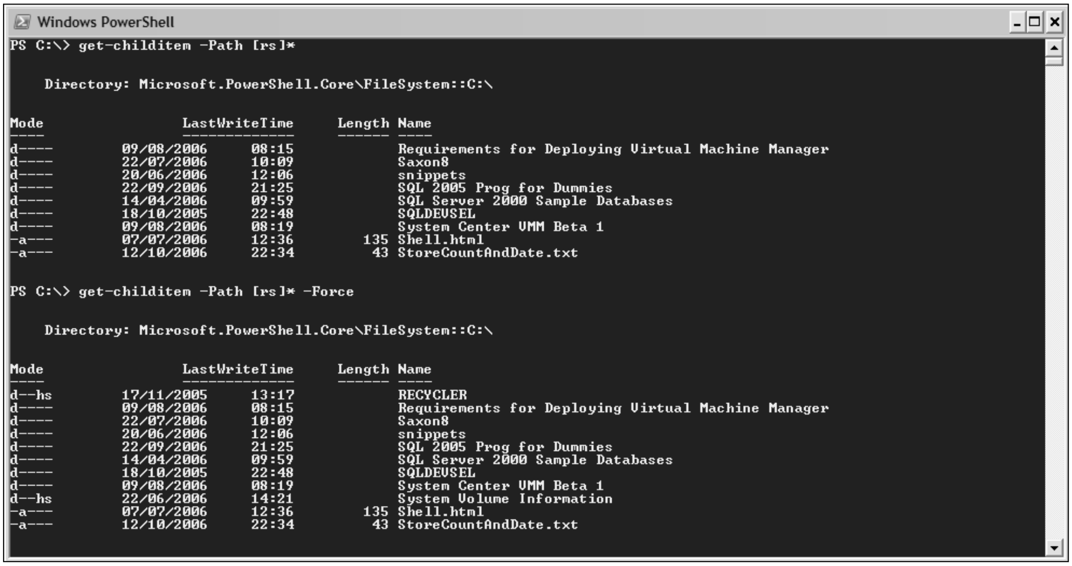


Figure 19-18

Tab Completion

Tab completion is available to you when using the `get-childitem` cmdlet and other cmdlets. As you can see in Figure 19-19, when I run the command:

```
get-childitem [pw]* |
where-object {$_.Mode -match "d"}
```

in the root directory of a Windows XP machine's hard disk, several folders are displayed, including the Windows folder and the Pro PowerShell folder.

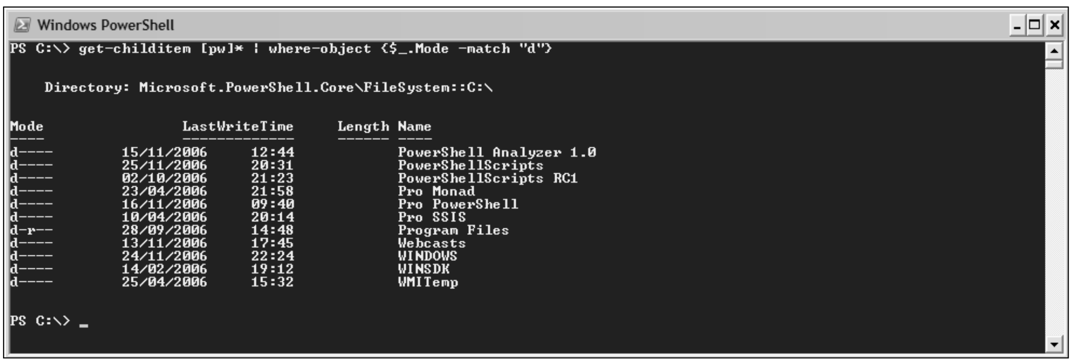


Figure 19-19

If you type

```
cd wi
```

then press the Tab key, the name of the single matching folder `C:\Windows` is completed for you.

Similarly, if you delete the preceding and type

```
cd Pro
```

then press the Tab key twice (since I still have a Pro Monad folder) the command changes cycles to

```
cd 'Pro PowerShell'
```

with the paired apostrophes intelligently added for you. If you intended to change the folder to the `C:\Program Files` folder, press Tab twice more and the command changes to:

```
cd 'Program Files'
```

Thus, Windows PowerShell will complete folder names for you. If there is ambiguity, pressing the Tab key additional times allows you to cycle through the matching options.

> *Be aware that in PowerShell 1.0 that tab completion for* `set-location` *or its* `cd` *alias also cycles through filenames. You can't set the location to be a file. I assume that this behavior will be corrected in a future version of PowerShell.*

Tab completion also works for each step of a multistep change in location. For example, to move to the folder `C:\Pro PowerShell\Chapter 19` (assuming it exists on the machine) from the root directory of drive `C:`, you can use the following commands. Type

```
cd Pro
```

then hit the Tab key. The command now reads:

```
cd 'Pro PowerShell'
```

Use the left arrow key to move the cursor to immediately before the right apostrophe, type `\Ch` and press Tab. The command now shows

```
cd 'Pro PowerShell\Chapter 19'
```

assuming that the only folder in the `Pro PowerShell` folder beginning with `Ch` is the `Chapter 19` folder. If you have other folders such as `Chapter 18` in the `Pro PowerShell` folder, you will need to press Tab multiple times until the preceding command is displayed.

Redirection

Windows PowerShell has two redirection operators, > and >>. A redirection operator redirects the output of a command (or pipeline) to a specified location. The > operator creates a new file and redirects text to it or, if the file exists, it overwrites the existing content. The >> operator appends text to an existing file without overwriting the existing content.

You can redirect text from the command line to a file. For example, to redirect the literal text Hello world! to a not yet existent file `NonExistent.txt` in the same directory use this command:

```
"Hello world!" > NonExistent.txt
```

You can check that the content has been added to the newly created file using the command:

```
get-content NonExistent.txt
```

If you then redirect new text to the file using the > operator, it overwrites the existing content, as you can demonstrate using the following commands:

```
"This overwrites the old text." > NonExistent.txt
get-content NonExistent.txt
```

But, if you use the >> operator, you can append text to the file, as you can demonstrate using the following commands:

```
"This appends a new line to the file." > NonExistent.txt
get-content NonExistent.txt
```

Figure 19-20 shows the results of executing the preceding commands in this section.

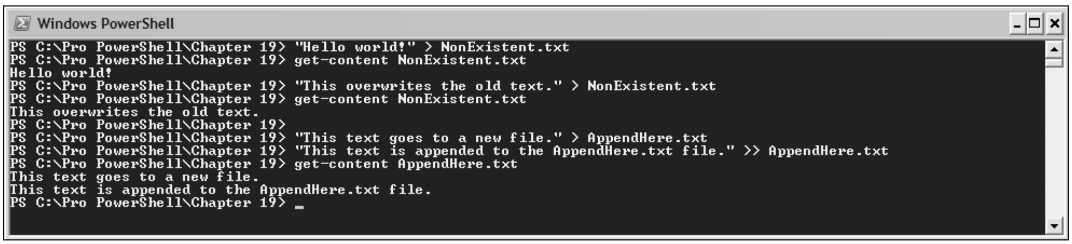


Figure 19-20

Similarly, you can use the >> redirection operator to append text to an existing file, as shown in Figure 19-20 when you execute the following commands:

```
"This text goes to a new file." > AppendHere.txt
"This text is appended to the AppendHere.txt file." >> AppendHere.txt
get-content AppendHere.txt
```

You can also redirect output from a pipeline to a file. The following script displays a timestamp together with the name and handle count on processes whose name begins with s:

```
"This script was executed: $(get-date)"

get-process s* |
format-table processname, handlecount -auto
```

The first line of the script executes the `get-date` cmdlet to provider a live date. By omitting the `write-host` cmdlet the text is echoed to the screen when you execute the script on its own, but the line can be redirected to a file when you add a redirection operator.

The following command executes the script `SimpleScript.ps1` and sends the output to the file `ProcessesWithTimestamp.txt`:

```
.\SimpleScript.ps1 > ProcessesWithTimestamp.txt
```

Confirm that the file has been created by using the following command. Notice the length of the file.

```
get-content ProcessesWithTimestamp.txt
```

If you want to only store one copy of information about running processes, then simply repeat the first command at some future time. However, if you want to append updated information to the file on multiple occasions use the >> redirection operator, as in the following command:

```
.\SimpleScript.ps1 >> ProcessesWithTimestamp.txt
```

As you can see in Figure 19-21, the size of `ProcessesWithTimestamp.txt` has approximately doubled, indicating that additional information about running processes has been appended.

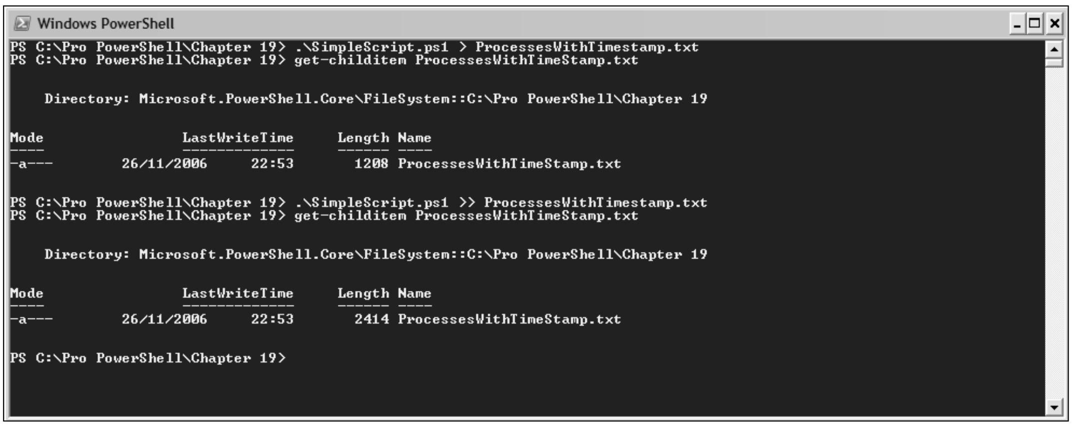


Figure 19-21

Creating Custom Drives

If you are working on the command line, it can be time-consuming, tedious, and error prone to type something like

```
cd "c:\My Documents\Test Scripts\PowerShell Book"
```

time after time. Tab completion makes the process easier, but you still have to use the Tab key and arrows keys Windows PowerShell allows you to create a custom drive, let's call it `book`, so that you can simply type

```
cd book:
```

and you will be in the directory that you want to be in.

In this example, I will show you how to create a custom drive. First, you will use Windows PowerShell to create a directory structure. To do that, use these commands (assuming that your current location is in the root folder of drive `C:`.

```
new-item "My Documents" -Type Directory
cd "My Documents"
new-item "Test Scripts" -Type Directory
cd "Test Scripts"
new-item "PowerShell Book" -Type Directory
cd "PowerShell Book"
```

As you can see in Figure 19-22, information about each new folder is displayed as it is created.

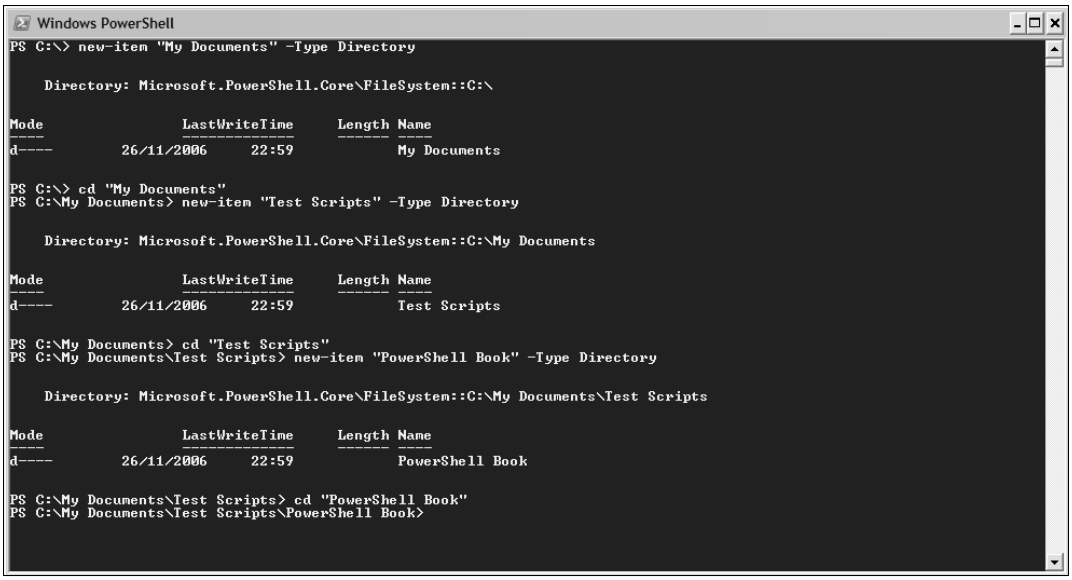


Figure 19-22

To create the new drive called `Book` and display some of its properties, use the following command:

```
new-psdrive -Name Book -PSProvider FileSystem -Root "C:\My Documents\Test
Scripts\PowerShell Book" |
format-list
```

Be careful not to include a colon character in the drive name in the preceding command. The value of the `Provider` parameter specifies that the `FileSystem` provider is used with the `Book` drive. The value of the `Root` parameter specifies the directory that is the root of the `Book` drive.

Navigate to the new drive using this command (this time you do need to include the colon character):

```
cd Book:
```

Create a file in the root folder of the new drive using this command:

```
"This is in drive Book:" > New.txt
```

Switch back to drive C: then navigate to the `C:\My Documents\Test Scripts\PowerShell Book` folder using these commands:

```
cd c:
cd "My Documents\Test Scripts\PowerShell Book"
```

Then you can confirm that the file that was added to the drive `Book` was added in the folder `C:\My Documents\Test Scripts\PowerShell Book` using this command:

```
get-childitem *
```

Figure 19-23 shows the results of running the preceding commands.

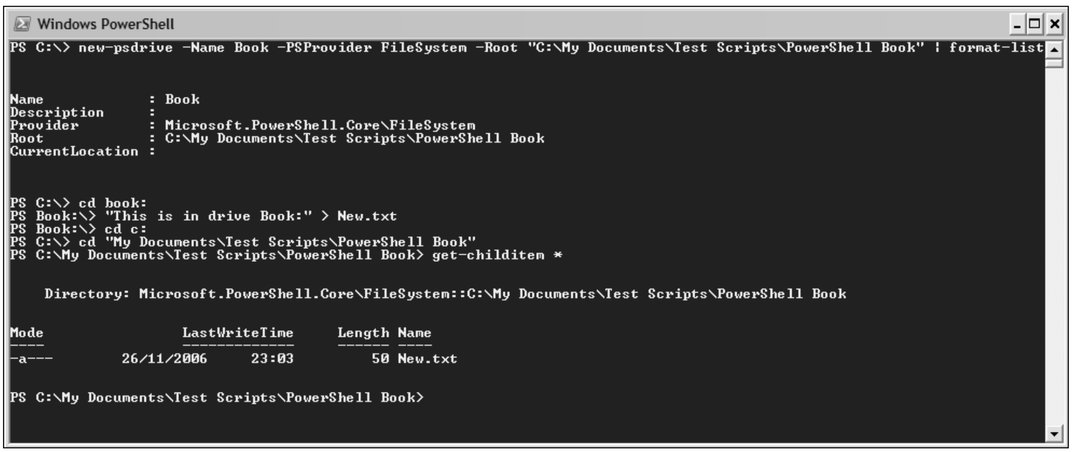


Figure 19-23

Cmdlets for File Actions

There are several cmdlets in Windows PowerShell version 1 that allow you to work with folders and files in the file system.

Using the out-file Cmdlet

The `out-file` cmdlet allows you to send the output of a command to a file. It is typically used as a step in a pipeline.

In addition to supporting the common parameters the `out-file` cmdlet supports the following parameters:

- `FilePath` — Specifies the path to the file where the command output is to be written. This parameter is a required positional parameter in position 1.
- `Encoding` — Specifies the encoding to be used when writing the file. This parameter is an optional positional parameter in position 2.
- `Append` — Specifies that data is to be appended to a file. This parameter is a named parameter.
- `Width` — Specifies how wide the individual lines of output are to be. A named parameter. The default value is 80.
- `NoClobber` — If present, specifies that an existing file will not be overwritten.
- `InputObject` — Specifies the input object. An optional parameter.

The following command outputs a table containing the names of running services to a file named `RunningServices.txt`. Since the file does not exist before the command is run, the file is created.

```
get-service -ServiceName * |
where-object {$_.status -eq "running"} |
format-table ServiceName, Status |out-file -filePath "C:\Pro PowerShell\Chapter
19\RunningServices.txt"
```

Figure 19-24 shows part of the content of `RunningServices.txt`.

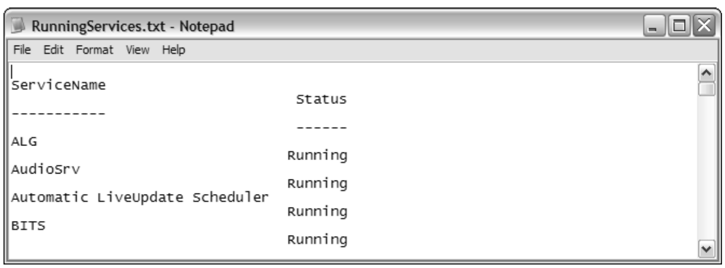


Figure 19-24

The following command uses the `get-date` cmdlet to get the current date and time. That datetime value is then output using the `out-file` cmdlet with the `Append` parameter set. The file `RunningServices.txt` already existed, having been created by the preceding command, so the `-append` parameter needs to be specified to allow data to be appended to the file.

```
get-date |
out-file -filePath "C:\Pro PowerShell\Chapter 19\RunningServices.txt" -append
```

As you can see in Figure 19-25, the date and time is added to the file.

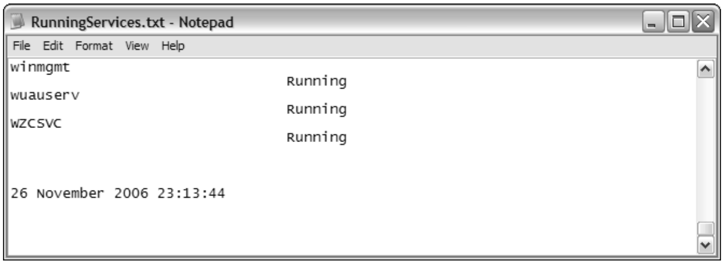


Figure 19-25

Using Cmdlets to Work with Paths

PowerShell provides cmdlets designed to let you work with paths:

- `convert-path`
- `join-path`
- `resolve-path`
- `split-path`
- `test-path`

These cmdlets can be used with PowerShell providers other than the `FileSystem` *provider. The following descriptions relate to their use with the* `FileSystem` *provider.*

The `test-path` cmdlet allows you to test if all elements of a path exist. In addition to the common parameters, it supports the following parameters:

- `path` — Specifies the path to be tested.
- `pathType` — Specifies the type of element that the path locates. Permitted values are `Container` (a folder in the `FileSystem` provider), `Leaf` (a file in the `FileSystem` provider), and `Any`. The default value is `Any`.
- `include` — Qualifies the value of the `path` parameter.

- `exclude` — Qualifies the value of the `path` parameter.
- `isValid` — Tests only whether the syntax of the path is valid. If present the existence of the path is not tested.
- `Filter` — Specifies a filter to apply when retrieving objects.
- `Credential` — Specifies a credential to get access to a resource.

The following command:

```
test-path "C:\Pro PowerShell\Chapter 19"
```

tests for the existence of the folder `C:\Pro PowerShell\Chapter 19`. In Figure 19-26, you can see that the folder exists and the command returns `True`.

At the time of running the following command:

```
test-path "C:\Pro PowerShell\Chapter 20"
```

the Chapter 20 folder had not been created, so the command returns `False`. Even though the folder doesn't exist, by using the `isValid` parameter you can test that the syntax of the path is valid.

```
test-path "C:\Pro PowerShell\Chapter 20" -isValid
```

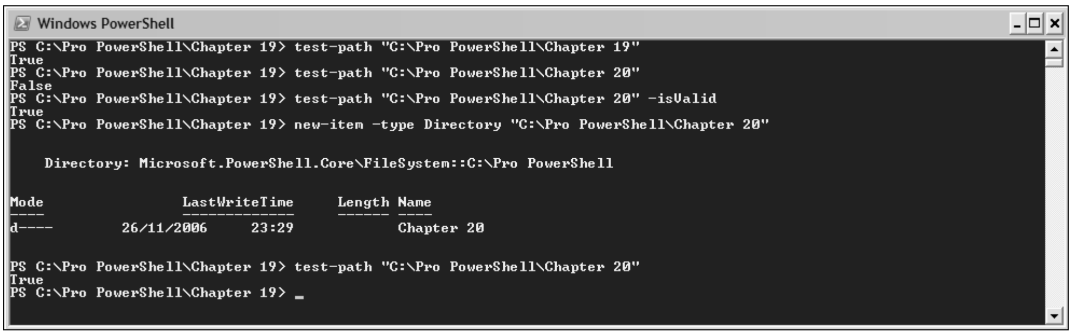


Figure 19-26

If you create the Chapter 20 folder:

```
new-item -type Directory "C:\Pro PowerShell\Chapter 20"
```

you can then confirm its existence by executing the following command:

```
test-path "C:\Pro PowerShell\Chapter 20"
```

The `join-path` cmdlet allows you to join a container portion of a path to a child path. The `join-path` cmdlet supports the following parameters in addition to the common parameters described in Chapter 6:

- `path` — Specifies the container (or main) portion(s) of a path. This is a positional parameter in position 1.
- `childPath` — Specifies the element to append to the value of the `-path` parameter. This is a positional parameter in position 2.
- `resolve` — Displays the items referenced by a joined path.
- `credential` — Specifies a credential to get access to a resource.

The following command joins two elements of a path that references `.txt` files in the `Pro PowerShell\Chapter 19` folder and displays the items it resolves to:

```
join-path -path "C:\Pro PowerShell\Chapter 19" -childPath "\Test*.txt" -resolve
```

As you can see in Figure 19-27 it resolves to three `.txt` files.

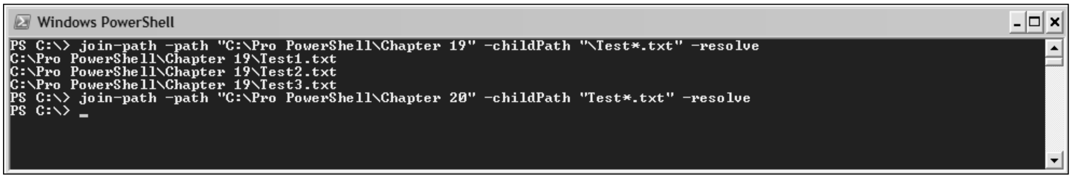


Figure 19-27

If you use a literal value for the `-path` parameter, be careful that you supply a `\` character to appropriately separate the components of the path. The following command omits the `\` character. As you can see in Figure 19-27, it fails.

```
join-path -path "C:\Pro PowerShell\Chapter 19" -childPath "Test*.txt" -resolve
```

If you use a wildcard in the value of the `-path` parameter, you don't need to be so careful about supplying a `\` character. For example, in the following command no `\` character is supplied but it works.

```
join-path -path C:\Win* -childPath System* -resolve
```

Figure 19-28 shows the result of executing the preceding command.

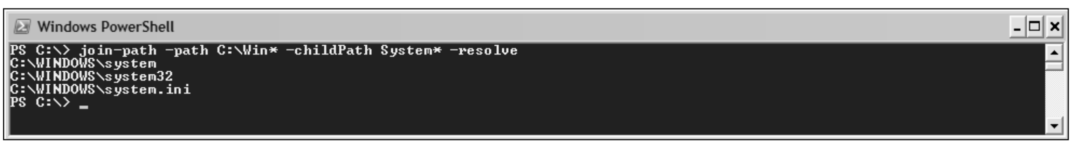


Figure 19-28

The resolve-path cmdlet resolves wildcard characters in a path and displays the items that the wildcards resolve to. In addition to the common parameters, the resolve-path cmdlet supports the following parameters:

- path — The path to be resolved. Wildcards are allowed. The parameter is positional in position 1.
- literalPath — The value is interpreted literally. Wildcards are not permitted.
- credential — Specifies a credential to get access to a resource.

The following command illustrates how to use the resolve-path cmdlet.

```
resolve-path -path "C:\Pro PowerShell\Chapter 19\Test*.txt"
```

Summary

In Windows PowerShell, you can use fully qualified or relative path names.

When using paired quotation marks or apostrophes with paths that include spaces, you need to use the & character if you intend that the command specified in a path is to be executed.

The get-psdrive cmdlet allows you to explore drives on a system. The get-childitem cmdlet allows you to explore folders and files on a system.

Windows PowerShell supports the creation of custom drives, to increase the convenience of access to frequently used folders with lengthy paths.

PowerShell supports several cmdlets to allow you to work with paths, including the test-path, join-path, and resolve-path cmdlets.

20

Working with the Registry

Windows PowerShell provides several *command shell providers* that allow you to work with data stores in a similar way to the ways you can work with the file system when using `CMD.exe`. By using a familiar file system metaphor, you should be able to navigate effectively in other hierarchical data stores without difficulty, assuming that you understand the structure of the store. Among the data stores that Windows PowerShell allows you to access in this way are the HKLM (`HKey_Local_Machine`) and HKCU (`HKey_Current_User`) hives of the Windows registry.

Windows PowerShell provides cmdlets to allow you to explore two registry hives and to alter the values held in registry keys. This functionality is powerful and flexible but, as with everything relating to the registry, you need to proceed with caution. If you make inappropriate changes to the registry, it is certainly possible to end up with a machine that won't run correctly or may not run at all. So be warned. Make changes to the registry only when you understand the implications of what you are doing. And check carefully for typos and other errors before you commit a change.

Introduction to the Registry

When an operating system boots up and while it's running, it needs to access pieces of information that indicate how the machine is configured to enable the operating system to start up and run. Since the introduction of Windows NT, the registry has been the store for such information in Windows operating systems. Previously, startup information was contained in a potentially large number of `.ini` files. As the number of files increased, performance potentially dropped off. The registry was introduced with a view to solving that problem and allowing a more coherent way to store startup and other configuration information.

The registry is a hierarchical data store. It stores configuration information relating to users, hardware, and applications. The data in the registry is stored in binary files, so it isn't readily accessible using standard text-editing applications.

Microsoft provides two GUI tools, `RegEdit.exe` and `RegEdt32.exe`. In the past, there were differences in the behavior of the two tools. In Windows XP and Windows Server 2003 the tools are essentially the same. `RegEdt32.exe` is a program that runs `RegEdit.exe`.

To run the Registry Editor, click Start ➪ Run; then type `RegEdit` in the text box. The Registry Editor opens. The appearance may differ a little from that shown in Figure 20-1, depending on any recent use of the Registry Editor. If you recently accessed a specific key in the registry, you will likely be taken back to that most recently viewed key.

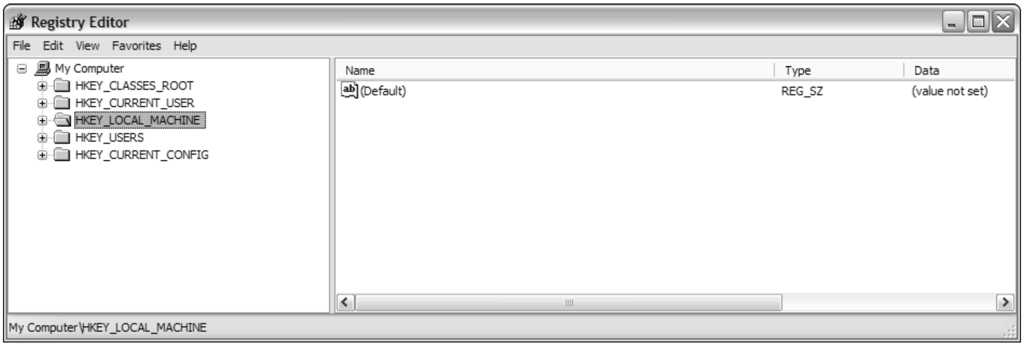


Figure 20-1

As you can see in Figure 20-1, there are five hives in the Windows registry:

- `HKEY_CLASSES_ROOT` — Ensures that the correct application opens if you click on a file in Windows Explorer; keeps track of file extensions and their associations with file types and programs. `HKEY_CLASSES_ROOT` is a subkey of `HKEY_LOCAL_MACHINE\Software`.
- `HKEY_CURRENT_USER` — Contains configuration for the current logged on user, including the user's folders, screen resolution and color settings, and Control Panel settings.
- `HKEY_LOCAL_MACHINE` — Contains configuration settings for the local machine that apply to any user.
- `HKEY_USERS` — Contains configuration information for active user profiles. `HKEY_CURRENT_USER` is a subkey of `HKEY_USERS`, although it is displayed as a separate hive in the Registry Editor.
- `HKEY_CURRENT_CONFIG` — Contains information about the hardware profile used at system startup.

Windows PowerShell supports access to the `HKEY_CURRENT_USER`, and `HKEY_LOCAL_MACHINE` hives.

Before you change anything in the registry Microsoft recommends that you backup the registry and also take time to understand what you need to do to be able to restore a working registry.

At the time of writing, Microsoft has a Knowledgebase article on backing up and restoring the registry on Windows XP and Windows 2003 at `http://support.microsoft.com/kb/322756`, including links to related articles. The article includes detailed instructions about how to export selected registry subkeys to back them up and how to back up the whole registry.

In the Registry Editor, the visual metaphor is similar to the metaphor for folders and files in Windows Explorer. In the left pane, click on a + sign to expand a container. When you click on the name of an item in the left pane, any corresponding information is displayed in the right pane.

Since the registry is a database it has allowed types. These are summarized briefly in the following table.

Type	Data Type	Description
Binary value	REG_BINARY	Raw binary data.
DWORD value	REG_DWORD	Data represented by a 32-bit integer.
Expandable String value	REG_EXPAND_SZ	A variable-length data string.
Multi-string value	REG_MULTI_SZ	A string that contains multiple values separated by spaces, commas, or other characters.
String value	REG_SZ	A fixed-length string.
Binary value	REG_RESOURCE_LIST	Nested arrays designed to store a resource list for use by, for example, a hardware device driver. Displayed as hexadecimal in the Registry Editor.
Binary value	REG_RESOURCE_ REQUIREMENTS_LIST	Nested arrays designed to store a device driver's list of possible hardware resources.
Binary value	REG_FULL_RESOURCE_ DESCRIPTOR	Nested arrays used by a hardware device.
None	REG_NONE	Data with no specified type. Displayed by the Registry Editor as hexadecimal.
Link	REG_LINK	A Unicode string naming a symbolic link.
QWORD value	REG_QWORD	Data represented by a 64-bit integer.

Many applications have keys whose values are of the type `REG_DWORD` (32-bit integer) or `REG_SZ` (fixed-length string). Figure 20-2 shows the keys for the Notepad application. Since Notepad can be configured for each user, it is found in the `HKEY_CURRENT_USER` hive.

Figure 20-2 shows how you can add a new key, using the Registry Editor.

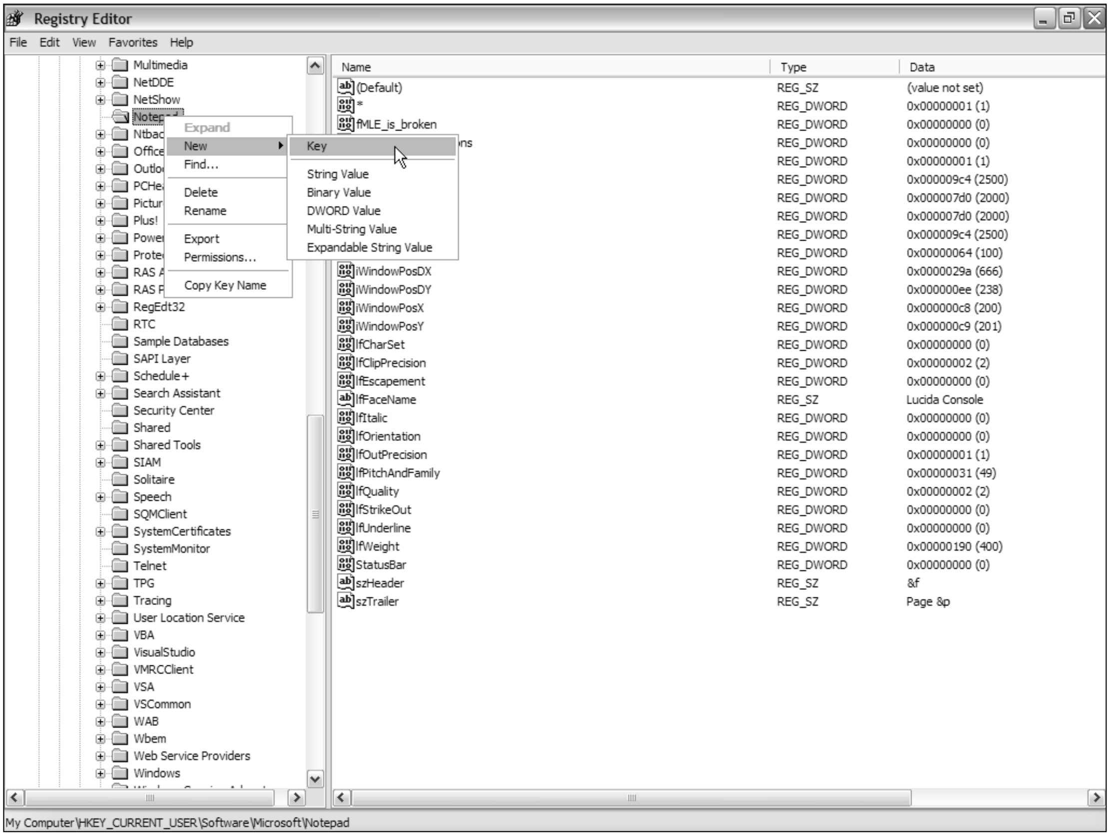


Figure 20-2

Exploring the Registry Using Windows PowerShell

Windows PowerShell makes it relatively straightforward to navigate the two supported registry hives. It allows you to select a registry hive as if it were a drive and then navigate around the hierarchy in the chosen hive as if you were navigating a hierarchy of folders and files.

Selecting a Hive

Navigating to a selected hive is straightforward. To navigate to the HKLM hive, use this command:

```
set-location HKLM:
```

or:

```
cd HKLM:
```

To navigate to the HKCU hive, use this command:

```
set-location HKCU:
```

or:

```
cd HKCU:
```

It's important that you include the colon character after the drive name. If you don't, then you'll receive an error message, as shown in Figure 20-3.

```
Set-Location : Cannot find path 'HKCU:\HKLM' because it does not exist.
At line:1 char:3
+ cd  <<<< HKLM
```

In the absence of a colon character, what you intend as a drive name is interpreted as a path relative to the current drive.

Figure 20-3

Navigating to a Desired Key

Information about how Windows PowerShell is configured to run (or not) scripts is contained in the `HKEY_LOCAL_MACHINE\Software\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell` key. As you can see in Figure 20-4, this includes the execution policy for Windows PowerShell scripts, set on this particular development machine, to `RemoteSigned`. Notice that the location of the `PowerShell.exe` file is also set there.

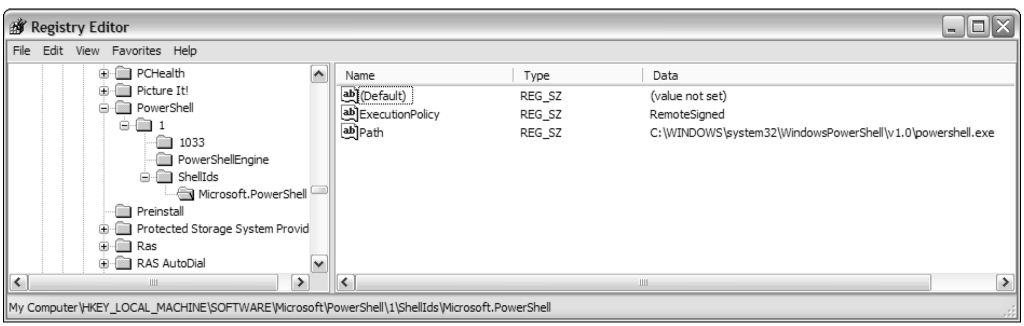


Figure 20-4

To display the value of the `ExecutionPolicy` key, follow these commands. They will work wherever you start from using one of the Windows PowerShell providers.

First navigate to the root of the `HKLM` drive:

```
set-location HKLM:\
```

Next navigate to the `ShellIds` key:

```
set-location Software\Microsoft\PowerShell\1\ShellIds
```

Finally, retrieve all the properties:

```
get-itemproperty *
```

The result of executing the preceding commands is shown in Figure 20-5. Notice that some of the displayed properties have a `PS` prefix and are PowerShell-specific. Other properties, such as `ExecutionPolicy` and `Path`, correspond to the properties you saw in the Registry Editor in Figure 20-4.

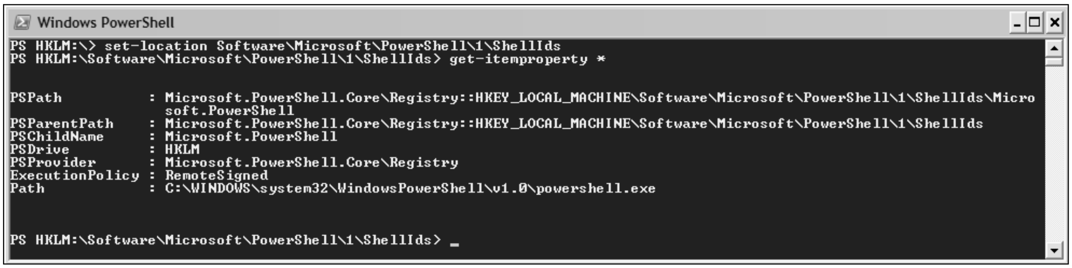


Figure 20-5

There is a seeming inconsistency in the values for the execution policy, but this is due to the value being set both for `ShellIds` and for the `Microsoft.PowerShell` keys. If you use the `get-itemproperty` cmdlet with the following fully qualified paths, you can see that the value for `ExecutionPolicy` is different in the two locations:

```
get-itemproperty HKLM:\Software\Microsoft\PowerShell\1\ShellIds\
Microsoft.PowerShell
```

and:

```
get-itemproperty HKLM:\Software\Microsoft\PowerShell\1\ShellIds
```

The `ExecutionPolicy` for `ShellIds` is `Restricted`, but this is overridden by the `ExecutionPolicy` for `Microsoft.PowerShell` (the console you typically see after installing Windows PowerShell), which is `RemoteSigned` on the machine whose data is shown in Figure 20-6.

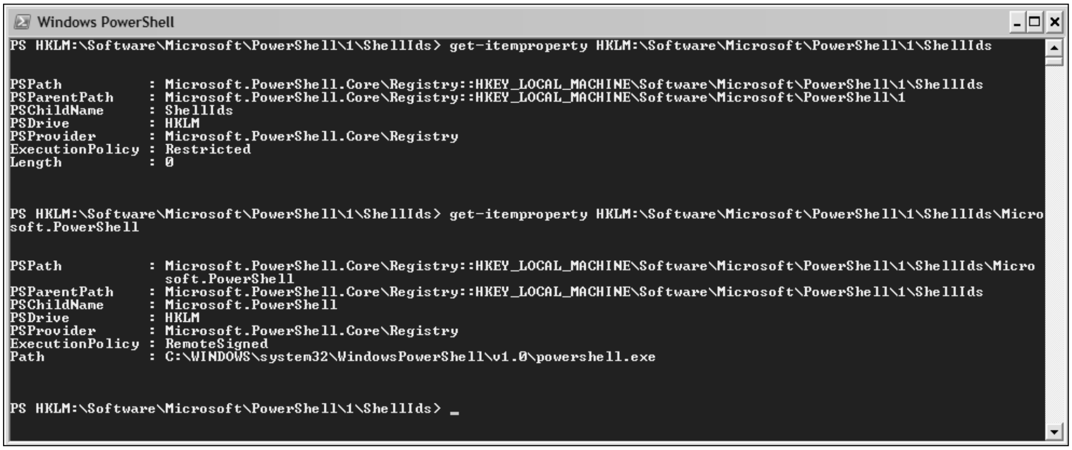


Figure 20-6

When you modify the execution policy using the `set-executionPolicy` cmdlet, it is the value for `ExecutionPolicy` at `HKLM:\Software\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell` that is changed. Similarly, the `get-executionPolicy` cmdlet retrieves the value at that location.

Changing the Registry

The first thing to say about changing *anything* in the registry is that you need to be sure that you know what you're doing. The registry is enormous, so becoming familiar with the meaning of registry keys and values takes a significant amount of time. If you don't understand the effect of a change you make in the registry, you are risking creating a machine that won't run correctly or at all. Please take note of the advice I gave you earlier in the chapter about making a registry backup before tinkering in the registry.

The following example changes the default behavior of Notepad on my machine from opening with no status bar to opening with a status bar. In Figure 20-2, the value for the `StatusBar` key is 0, meaning that Notepad opens with no status bar. The following command displays the same information:

```
get-itemproperty HKCU:\Software\Microsoft\Notepad -Name StatusBar
```

To set the value of `StatusBar` to 1, use the following command:

```
set-itemproperty HKCU:\Software\Microsoft\Notepad -Name StatusBar -Value 1
```

To confirm that the change has been made to the `StatusBar` property, run this command again:

```
get-itemproperty HKCU:\Software\Microsoft\Notepad -Name StatusBar
```

Launch Notepad and observe that a status bar is now displayed.

To set the value of `StatusBar` back to 0, use the following command:

```
set-itemproperty HKCU:\Software\Microsoft\Notepad -Property StatusBar -Value 0
```

You can see in Figure 20-7 the execution of the preceding commands. Launch Notepad again and observe that it now launches without a status bar.

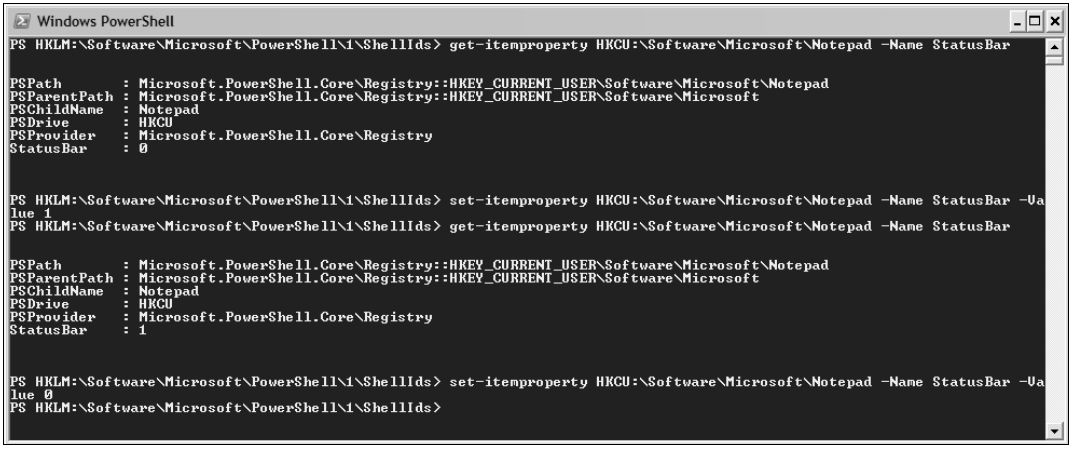


Figure 20-7

In Figure 20-8 I show two Notepad windows. The rear window was opened when the value of `StatusBar` was set to 1. You can see that the Notepad window has a status bar. The front Notepad was opened after I set the value of `StatusBar` back to 0. As you can see in Figure 20-8, that window has no status bar.

Figure 20-8

If you want to follow in the Registry Editor changes produced by Windows PowerShell be aware that those changes are not always reflected immediately in the values displayed in the Registry Editor. If you close and restart the Registry Editor, the changes you make from Windows PowerShell will be consistently displayed.

To add a dummy key to the Notepad key in `HKCU:\Software\Microsoft\Notepad`, execute these commands:

```
cd HKCU:\Software\Microsoft\Notepad
new-item DummyKey
```

Notice in Figure 20-9 that the information about the newly added key is displayed when the `new-item` cmdlet is executed.

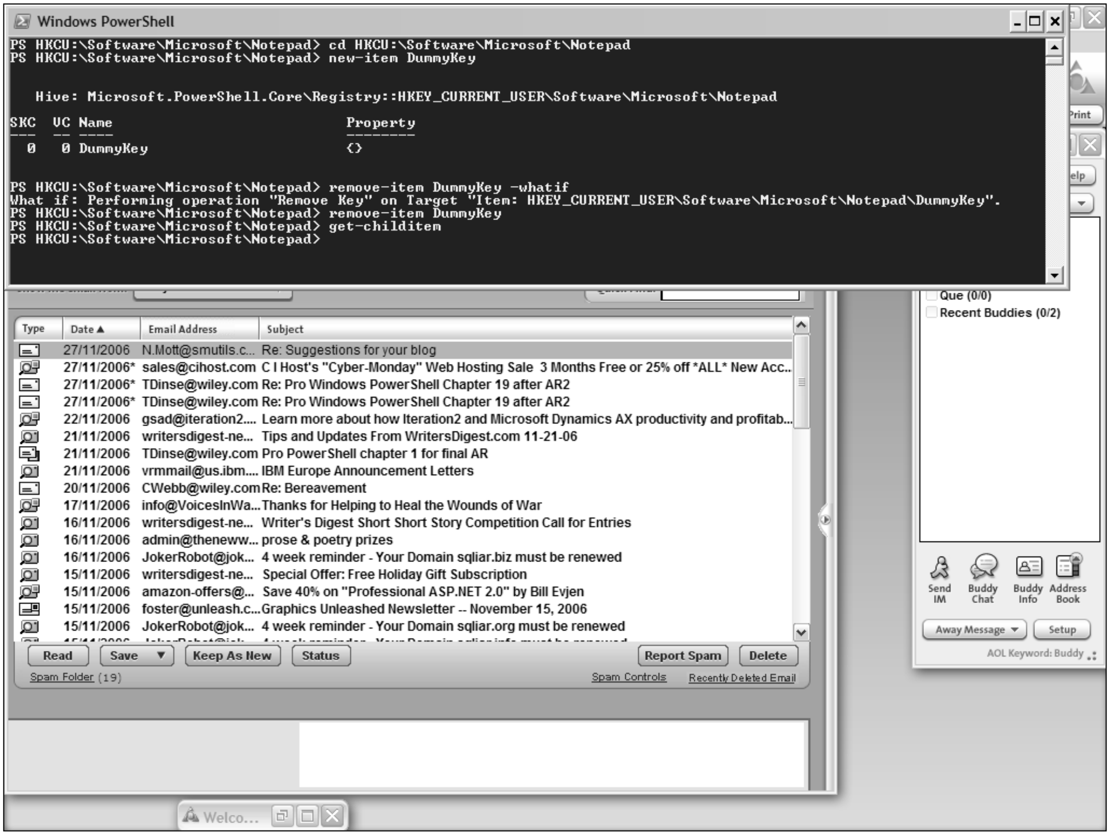


Figure 20-9

The `remove-item` cmdlet is used to delete keys. Use this with great care! You can use the `-whatif` parameter to see the effect of a command before executing it.

```
remove-item DummyKey -whatif
```

You can see in Figure 20-9 the information displayed when you execute the preceding command. It's also a good idea to use a fully qualified path to ensure that you know exactly what you are targeting.

To delete the `DummyKey` key added using the preceding commands, remove the `-whatif` parameter and execute the command. To confirm that the `DummyKey` key has been deleted, use the following command:

```
get-childitem
```

since the `DummyKey` key is a child of the `Notepad` key.

Once you appreciate how to explore properties in the registry and how to change and create keys and values, how you use Windows PowerShell to work with the registry depends on your knowledge of the registry keys and how you need to use them.

Summary

The Windows registry stores important information that supports the startup and running of machines with modern Windows operating systems. The Registry Editor allows you to inspect and manipulate keys and values in the registry.

You can use the `set-location` cmdlet to navigate around the registry. Use the `get-itemproperty` cmdlet to inspect values in a key. Use the `set-itemproperty` cmdlet to change values.

Use the `new-item` cmdlet to create a new key in the registry. Use the `remove-item` cmdlet to remove a key from the registry. Use the `remove-item` cmdlet with great care. Test out possible deletions by using the `-whatif` parameter to see what a command does before deciding whether or not to execute it.

21

Working with Environment Variables

Like the file system and registry, Windows PowerShell provides a command shell provider and corresponding drive that allows you to explore and manipulate environment variables. An environment variable is a value that can affect how the operating system or processes run.

Working with environment variables is simpler in some respects than working with the file system or the registry, since the environment variables are not stored in a hierarchy inside the `env` drive unlike the hierarchy of objects in the file system or registry. In the `env` drive, it is as if all Windows files were stored in the `root` folder. It is inappropriate, then, for example, to use the `Recurse` parameter when using the `get-childitem` cmdlet in the `env` drive that you might use with a drive associated with the `FileSystem` provider.

Another limitation when using the `env` drive is that you cannot make permanent changes in environment variables. Changes you make to environment variables are limited to the duration of the relevant MSH session and apply only to that session.

Environment Variables Overview

Environment variables are strings that contain information about a Windows system and/or about the configuration for the currently logged on user. This can affect how the operating system or individual processes behave. When Windows is being installed it configures environment variables, for example the path to the Windows files is contained in the `windir` environment variable. To see the value for the `windir` variable in Windows PowerShell, use the following command:

```
get-childitem var:windir
```

Alternatively, you can access the `windir` environment variable using this command:

```
$env:windir
```

As you can see in Figure 21-1, the value for the `windir` environment variable on my machine is `C:\Windows`.

Figure 21-1

To modify system environment variables, you must have administrator privileges. In addition to system environment variables such as `windir`, individual applications may create their own environment variables or users may specify environment variables.

To access information about environment variables using the Windows graphical user interface, select Start ➪ My Computer then right-click and select Properties. On the System Properties dialog box, select the Advanced tab, which is shown in Figure 21-2.

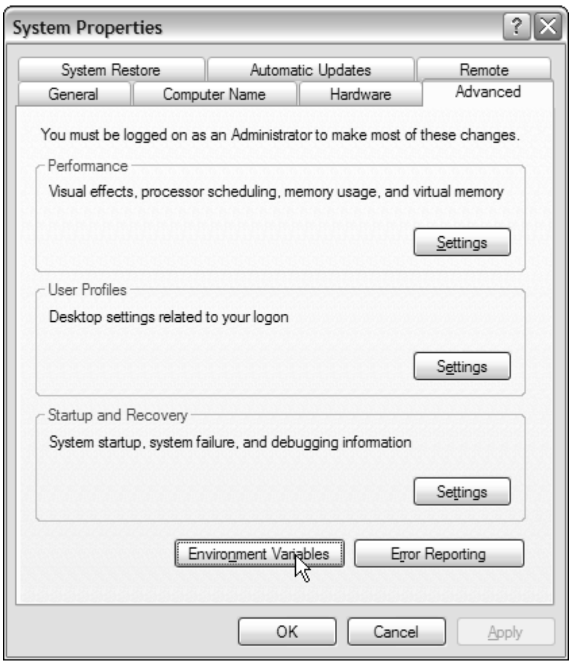


Figure 21-2

On the Advanced tab click the Environment Variables button, which is shown moused in Figure 21-2. The Environment Variables dialog box shown in Figure 21-3 opens.

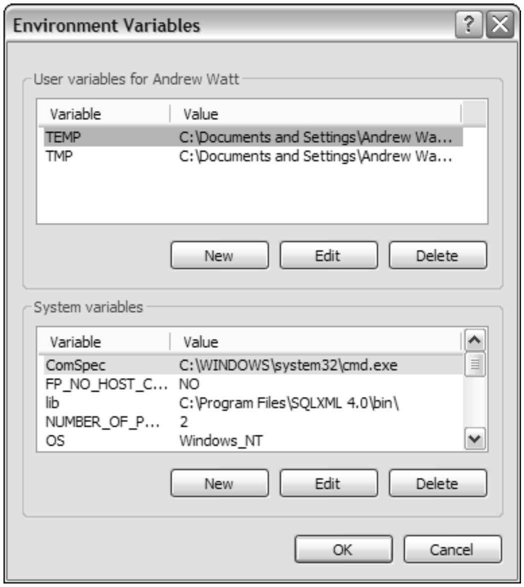


Figure 21-3

As you can see in Figure 21-3, the environment variables are divided into User Environment Variables and System Environment Variables. The User Environment Variables are specific to a particular user. The System Environment Variables apply to all users. Each category has buttons to allow a user to create a new variable, edit an existing variable, or delete an existing variable.

The interface to allow editing can be clumsy. For example, select the `PATH` system environment variable; then click the Edit button. The Edit System Variable dialog box shown in Figure 21-4 opens. Only part of the value of the `PATH` system environment variable is displayed, making it difficult to see the current value of the `PATH` variable and, particularly, when you need to add a lengthy path, it is also difficult to see the full path you have added to the variable's value.

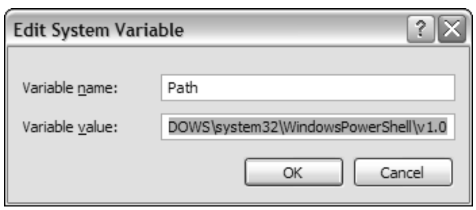


Figure 21-4

However, often the value for the PATH variable is very long, as you can see in Figure 21-5, where the value for the PATH environment variable is displayed in Windows PowerShell using the command:

```
get-childitem env:PATH |
format-list
```

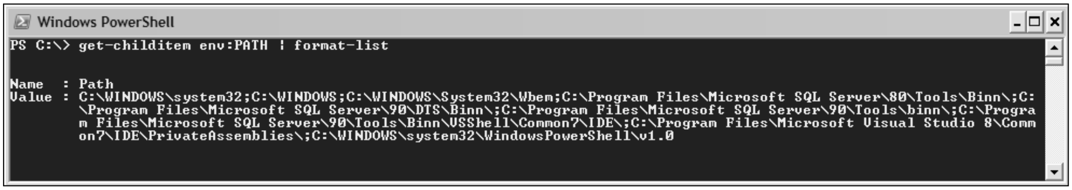


Figure 21-5

The way that the value of the PATH environment variable is displayed in Figure 21-5 isn't very readable either. Since the component parts of the value of the PATH environment variable are simply strings, you can use methods of the String class to display the individual components of the PATH environment variable and improve readability. Use the following command to display each folder on a separate line:

```
$env:PATH.ToString().Split(';')
```

When referencing an environment variable using the $ notation above you do not use a cmdlet. Conversely, when you supply the name of an environment variable as the value of a cmdlet's parameter, you do not use the $ sign before the environment variable's name.

The ToString() method converts the object returned by $env:PATH to a string. Then the Split() method of the String object is used to split the value of the PATH environment variable at each occurrence of a semicolon. The character which is the argument to the Split() method determines where the original string is split. In this case, the value of the PATH variable is split on the occurrence of a semicolon. Figure 21-6 shows the directories that make up the PATH environment variable displayed in a more readable way.

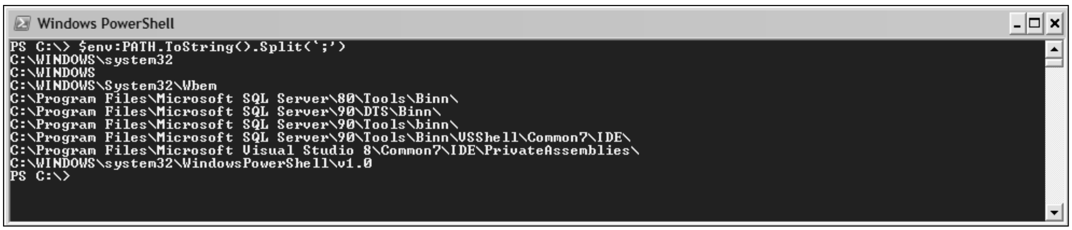


Figure 21-6

The Environment Command Shell Provider

Access to environment variables from Windows PowerShell depends on the Environment command shell provider. The Environment command shell provider is the interface between Windows PowerShell and the data store for environment variables. The `env:` drive containing environment variables is supported by the `Environment` command shell provider and is the only drive related to it. The `env` drive contains the information about all currently configured environment variables.

To display information about the `Environment` provider, use this command:

```
get-psprovider -PSProvider Environment |
format-list
```

The information displayed by the preceding commands is shown in Figure 21-7. As you can, see the Environment provider supports a single drive, `env`.

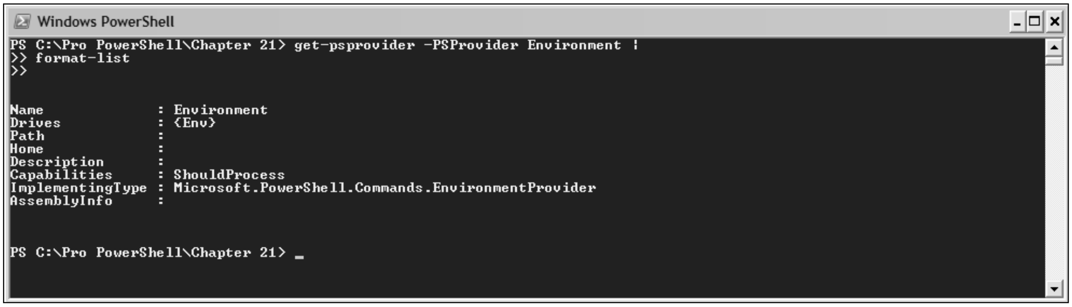


Figure 21-7

To display the current help file about the Environment provider, use this command:

```
Help Environment
```

To display information about the `Env` drive, use this command:

```
get-psdrive Env |
format-list
```

Although `Env` is a drive, be careful not to include a colon character in the preceding command after the drive name. Figure 21-8 shows the information displayed about the `Env` drive after executing the preceding command.

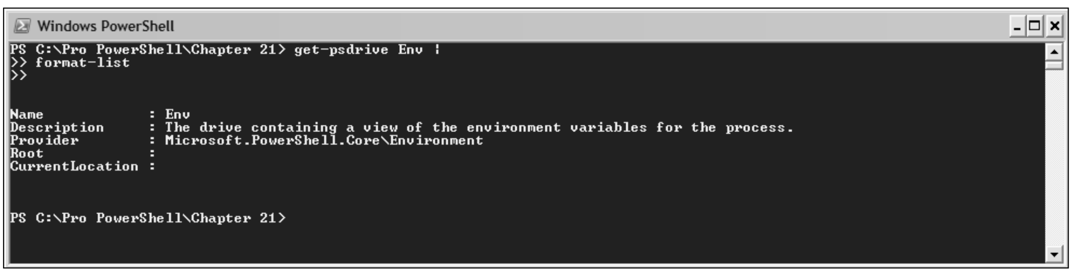


Figure 21-8

Exploring Environment Variables

Exploring environment variables is straightforward using the `get-childitem` cmdlet. To see a complete list of environment variables, use this command from any location:

```
get-childitem env:*
```

To sort the environment variables alphabetically and then page the output, use this command:

```
get-childitem env:* |
sort-object Key |
more
```

The `sort-object` cmdlet in the second step of the pipeline sorts the objects passed to it by the first step of the pipeline in ascending alphabetical order by the value of the `Key` property. Each environment variable is returned as a `System.Collections.DictionaryEntry` object. That object's `Key` and `Value` properties are the most likely to be of interest to you.

Figure 21-9 shows one screen of the environment variables on a Windows XP SP2 machine.

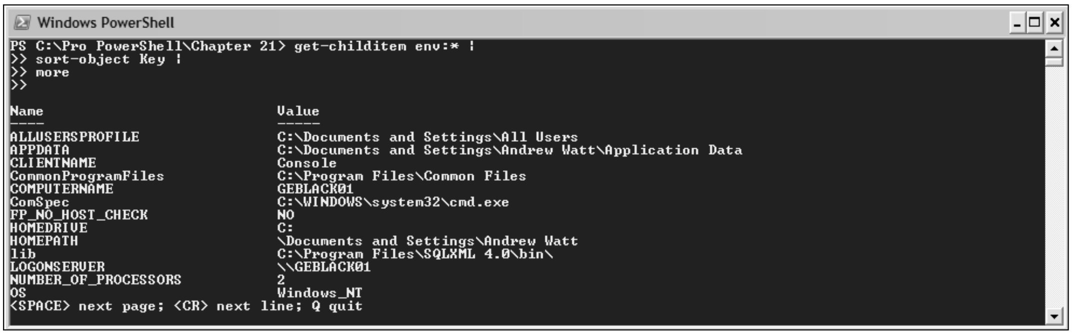


Figure 21-9

Alternatively, to display environment variables and navigate to the `env` drive, use the `get-childitem` cmdlet:

```
set-location env:
get-childitem * |
sort-object Key |
more
```

An alternative form of the second command that produces the same results is:

```
get-childitem * |
sort-object {$_.Key} |
more
```

You can display information about the environment variables relating to processors using this command, assuming you are already in the `env` directory:

```
get-childitem *process*
```

Figure 21-10 shows the results of executing the preceding command.

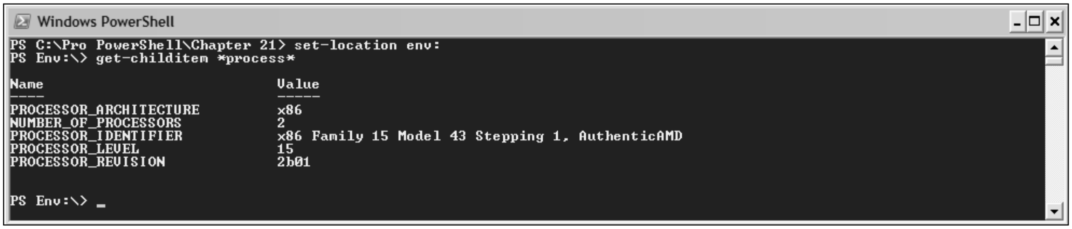


Figure 21-10

To find the properties of environment variables, which are `DictionaryEntry` objects, use this command:

```
get-childitem |
get-member
```

Notice in Figure 21-11 that there is an `AliasProperty` called `Name`, so that you can use the alias `Name` for the `Key` property. Each environment variable is essentially a key-value pair, as indicated by the exposed properties.

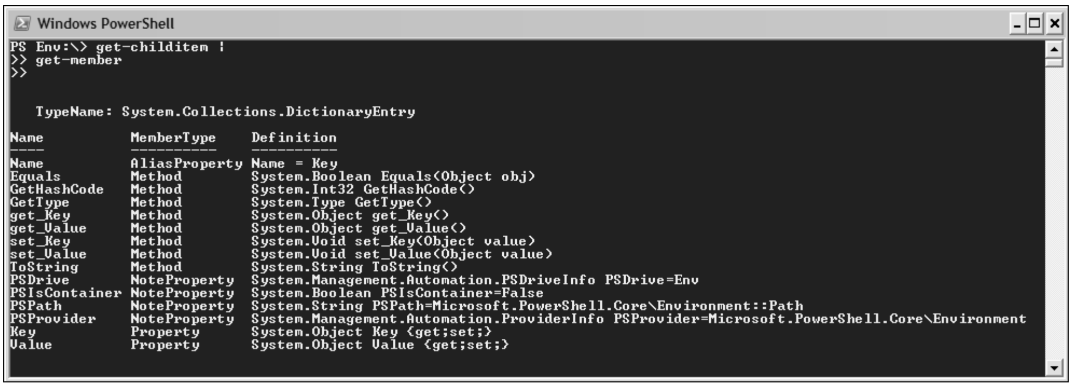


Figure 21-11

Modifying Environment Variables

You can modify environment variables but only for the duration of a Windows PowerShell session. The following example adds a new directory `C:\` to the `PATH` environment variable.

You can display the folders in the `PATH` environment variable using the following command:

```
$env:PATH.ToString().Split(';')
```

Then assign the current path to the variable `$OriginalPath`, using this command:

```
$OriginalPath = $env:PATH
```

Then a concatenation of `$OriginalPath` and the literal string `";C:\"` is assigned to the `PATH` environment variable. The reason for including the semicolon as the first character of the additional path is that there is no terminating semicolon on the original path. If the value of the `PATH` environment variable already is a semicolon, it's not necessary to have a semicolon as the first character of the additional path.

```
$env:PATH = ($OriginalPath + ";C:\")
```

Finally, I display the new value of the `PATH` environment variable. Notice in Figure 21-12 that the newly added directory `C:\` is the final directory in the `PATH`.

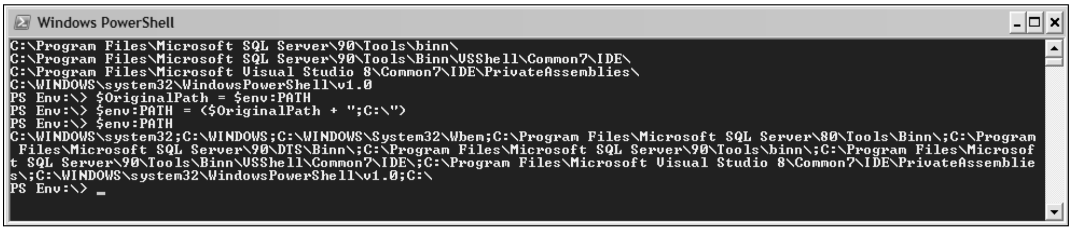


Figure 21-12

An alternative syntax to add a new path to the `PATH` environment variable is shown here:

```
$env:PATH += ";C:\"
```

The folders included in the `PATH` environment variable affect the syntax you use when running a PowerShell script. If the current directory is not part of the `PATH` environment variable, you need to use the syntax:

```
.\Scriptname
```

or:

```
.\Scriptname.ps1
```

Once you add a folder to the `PATH` environment variable, you can omit the initial period and backslash, as shown in this example. I created a simple script, `ShowDate.ps1`, which contains a single command:

```
get-date
```

and saved it in the folder `C:\`. In a new PowerShell session, the value of the `PATH` environment variable does not include `C:\`. So if you type

```
ShowDate
```

you see the following error message:

```
The term 'ShowDate' is not recognized as a cmdlet, function, operable program, or
script file. Verify the term and try
again.
At line:1 char:8
+ ShowDate <<<<
```

Next, add the folder `C:\` to the `PATH` environment variable using this command:

```
$env:PATH += ";C:\"
```

Since the script `ShowDate.ps1` is in a folder that is now part of the `PATH` environment variable, I can now run the script simply by typing:

```
ShowDate
```

As you can see in Figure 21-13, the script now executes and displays the current date and time.

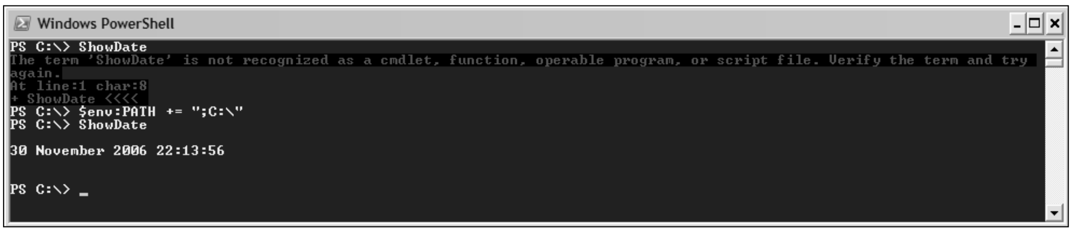


Figure 21-13

Modifying the value of `PATH` can be useful if you want to run one or more scripts from a particular folder but want to avoid the bother of typing the full path every time you want to run them.

You need to be keep in mind that any changes made to environment variables apply only to the current PowerShell session. If, for example, you open a new PowerShell window changes made in another PowerShell session have no effect. If you want to change the value of an environment variable routinely when you start PowerShell, then add an appropriate statement to a profile file.

Summary

The values of environment variables are exposed in the `env:` drive by the Windows PowerShell `Environment` provider.

To retrieve the value of environment variables, you can use the `get-childitem` cmdlet or use the variable syntax of the form `$env:variableName`.

To change the value of an environment variable for the duration of a PowerShell session, use the appropriate Windows PowerShell assignment operator.

Part III
Language Reference

Chapter 22: Working with Logs

Chapter 23: Working with WMI

22

Working with Logs

A common administrative task is checking or examining event logs. Event logs contain useful information about the execution of the Windows system, of applications on a machine and whether any security issues have occurred. The event logs have a series of categories (entry types) that indicate the significance of the event being logged.

Windows PowerShell version 1.0 provides one cmdlet that supports event logs: the `get-eventlog` cmdlet, which displays information from the local machine.

Event Log Basics

If you have spent any significant time working with Windows machines, you will likely have spent at least some time monitoring the behavior of applications on one or more machines and checked what errors are logged in time association with system or application malfunction. The GUI tool to support viewing of events is the Event Viewer. To launch Event Viewer, select Start ➪ Administrative Tools ➪ Event Viewer. Figure 22-1 shows the Event Viewer as seen on a Windows XP machine with Windows PowerShell installed. Depending on installed software, you may see additional logs displayed in Event Viewer.

If you had prerelease versions of Windows PowerShell installed, you may see other logs that use the term PowerShell or Monad. If you install the final release of Windows PowerShell 1.0 on a machine with no previous installation, the Windows PowerShell log is used. If you installed Release Candidate 2, the PowerShell log is used and the Windows PowerShell log is ignored.

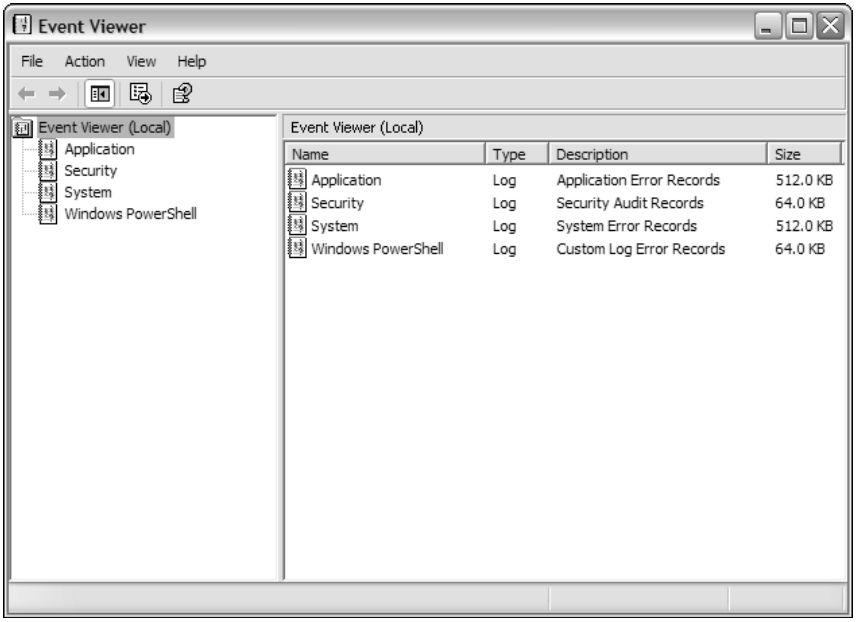


Figure 22-1

The Application, Security and System logs are routinely present on Windows XP and Windows 2003 machines.

To view the properties of a specified log, select it the left pane of the Event Viewer, then select Action ➪ Properties. The properties dialog box for the selected event log is displayed. Figure 22-2 shows the properties dialog box for the Windows PowerShell event log on a Windows XP machine.

The default behavior of the Windows PowerShell event log on Windows 2003 or Windows XP machine is a maximum log size of 15,360KB and overwriting of events as needed.

The Filter tab of the properties dialog for an event log specifies which events in the log are displayed in the Event Viewer for a given event log. Figure 22-3 shows the default appearance of the Filter tab.

To change the amount of data displayed for an event log, an administrator has to select Action ➪ Properties, then click the Filter tab then check or uncheck checkboxes corresponding to available event types. Additional selections can be made from the Event Source dropdown. Then yet a further selection can be made from the Category dropdown. If a particular time window is of interest, start and end times can be specified using the From and To dialogs. Figure 22-4 shows some of the available options for the category for an Application event log.

To use the Filter tab to apply filters, a user doesn't have to have extensive knowledge of the event log system. However, using the graphical interface can be slow and cumbersome if several filters are to be applied to give different views of the events that have been logged. A command line option to inspect event logs such as that provided by the Windows PowerShell `get-eventlog` cmdlet can be easier and quicker to use, at least by those who understand the cmdlet's syntax.

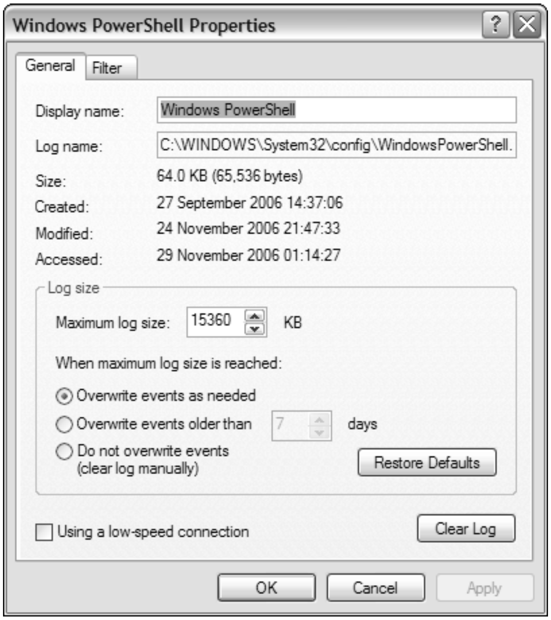


Figure 22-2

Figure 22-3

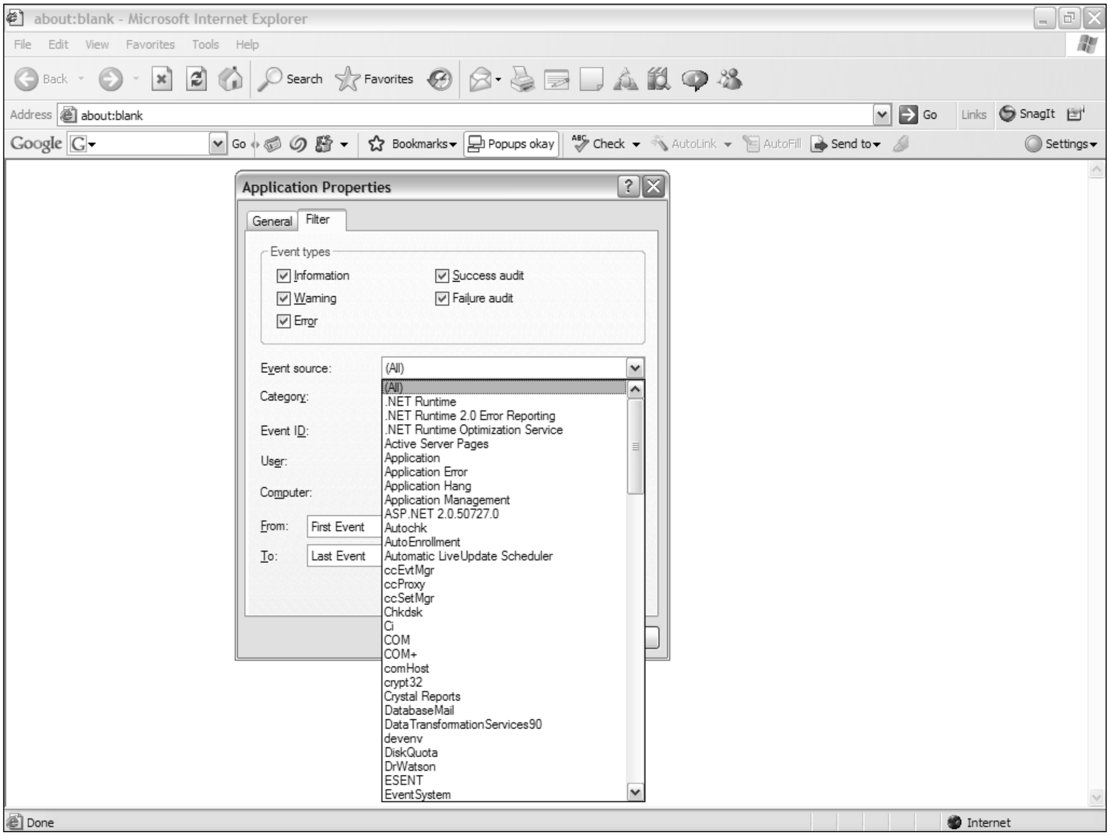


Figure 22-4

The get-eventlog Cmdlet

The `get-eventlog` cmdlet allows you to access information contained in various event logs or to list the event logs on the local machine.

In Windows PowerShell version 1.0 the `get-eventlog` *cmdlet can access only the local machine. Like other core Windows PowerShell functionality which doesn't explicitly use Windows Management Instrumentation the* `get-eventlog` *cmdlet is limited in scope to the local machine. It is likely that a later version of Windows PowerShell will support retrieval of event log information across a network.*

In addition to supporting the common parameters, the `get-eventlog` cmdlet supports the following parameters:

- `LogName` — The name of the log whose content is to be retrieved. This is a required parameter, which is a positional parameter in position 1. It does not support multiple values or wildcards.
- `Newest` — Specifies a number. That number represents how many entries are to be retrieved.

- `List` — Specifies a list of available event logs. This is a named parameter.
- `AsString` — Indicates that the entries in an event log are to be retrieved as string values rather than as objects. Can be used with the `-list` parameter.

A simple use of the `get-eventlog` cmdlet is to display the available event logs on the local machine. To do that, use this command:

```
get-eventlog -List
```

Figure 22-5 shows the event logs available on a Windows XP machine with Windows PowerShell installed, together with information about their maximum size the action to be carried out when the event log is full.

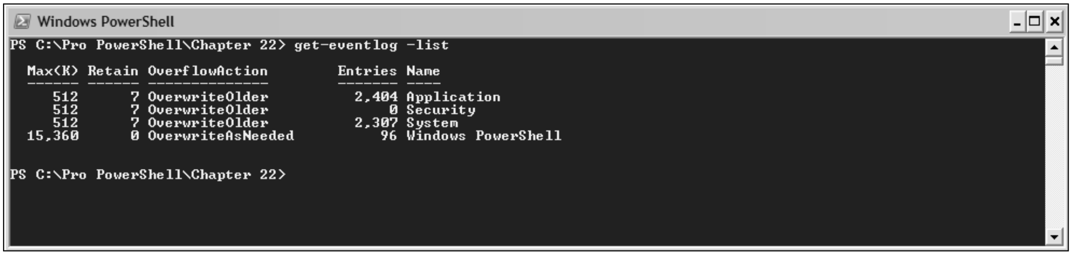


Figure 22-5

Fuller information on each available event log can be displayed by combining the preceding command with the `format-list` cmdlet in a simple pipeline:

```
get-eventlog -List |
format-list *
```

Figure 22-6 shows the information displayed for the Application and Security logs.

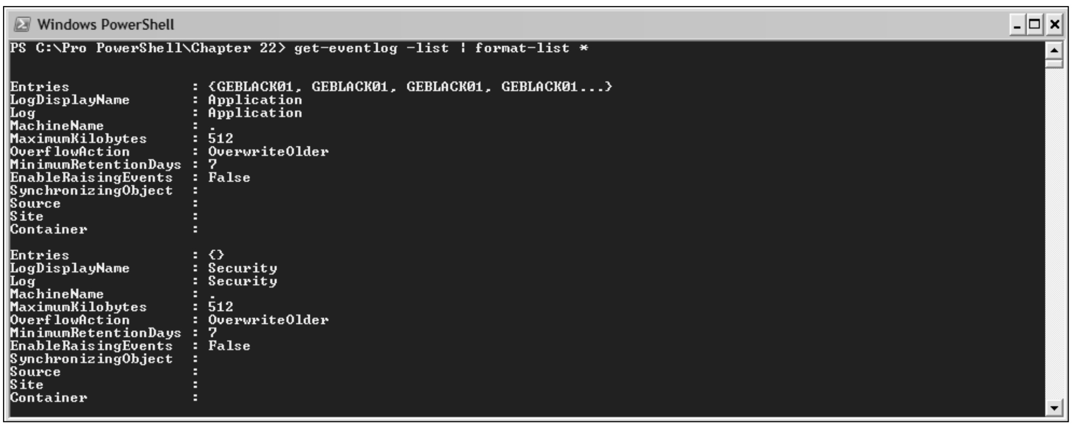


Figure 22-6

To retrieve all the information in a specified event log, use the `get-eventlog` cmdlet with the `-LogName` parameter. For example, to display all information in the Windows PowerShell event log, use this command. Notice that, since the name of the log includes a space character, that the name must be enclosed in quotation marks. Alternatively, use paired apostrophes:

```
get-eventlog -LogName "Windows PowerShell"
```

or, since the `-LogName` parameter is a positional parameter:

```
get-eventlog "Windows PowerShell"
```

In practice, it is often more convenient to use the preceding command combined with the `out-host` cmdlet to page the output:

```
get-eventlog -LogName "Windows PowerShell" |
out-host -paging
```

or, its functional equivalent:

```
get-eventlog -LogName "Windows PowerShell" |
more
```

Figure 22-7 shows a little of the content on a Windows PowerShell event log on a Windows XP machine.

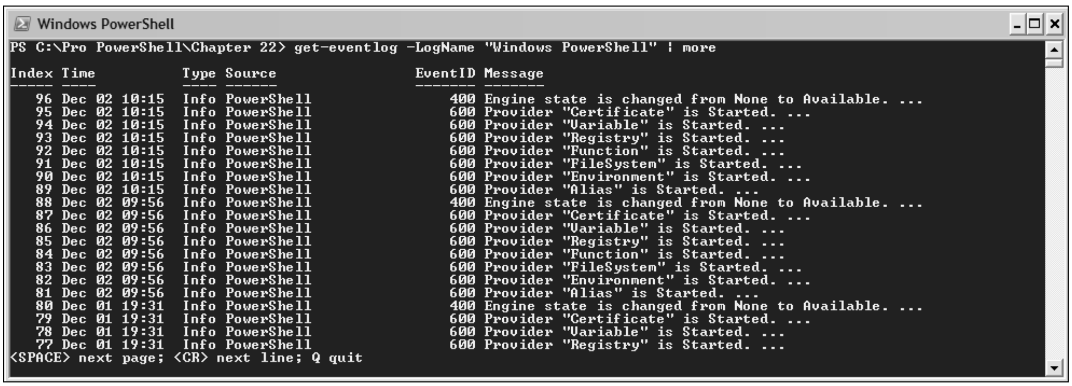


Figure 22-7

The large volume of information in a typical event log means that you have to filter the information in some way. One simple technique to explore the available options is to find the members of the event log of interest using the `get-member` cmdlet. For example, to find the members of the Windows PowerShell event log, use this command:

```
get-eventlog -LogName "Windows PowerShell" |
get-member
```

Figure 22-8 shows the properties returned by the preceding command.

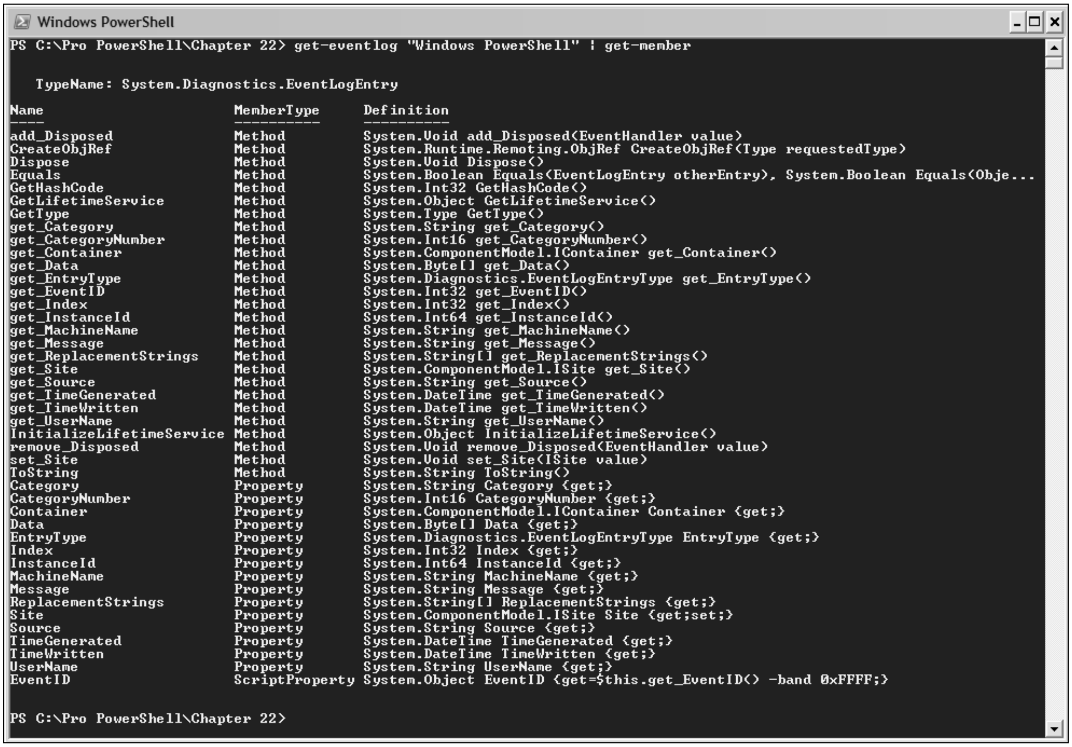


Figure 22-8

Notice in Figure 22-8 that several of the properties broadly correspond to the columns displayed in Event Viewer. However, the names of the properties exposed in Windows PowerShell don't correspond exactly to the characteristics exposed in the Event Viewer interface or the column headers in Windows PowerShell's default layout. For example, in Event Viewer, you see two distinct columns: Date and Time; in Windows PowerShell's default layout, the column header for both date and time is Time, and the property names are `TimeGenerated` and `TimeWritten`.

Use the `group-object` to get an overview of the content of an event log. For example, to find out which applications have events recorded in the Application event log, use this command:

```
get-eventlog -LogName Application |
group-object Source
```

The display of the results can be improved by adding the `format-table` cmdlet to the pipeline, as in the following command:

```
get-eventlog -LogName Application |
group-object Source |
format-table -auto
```

Figure 22-9 shows the result of running the preceding command. You can see that there is a lot of activity in the Application log relating to SQL Server.

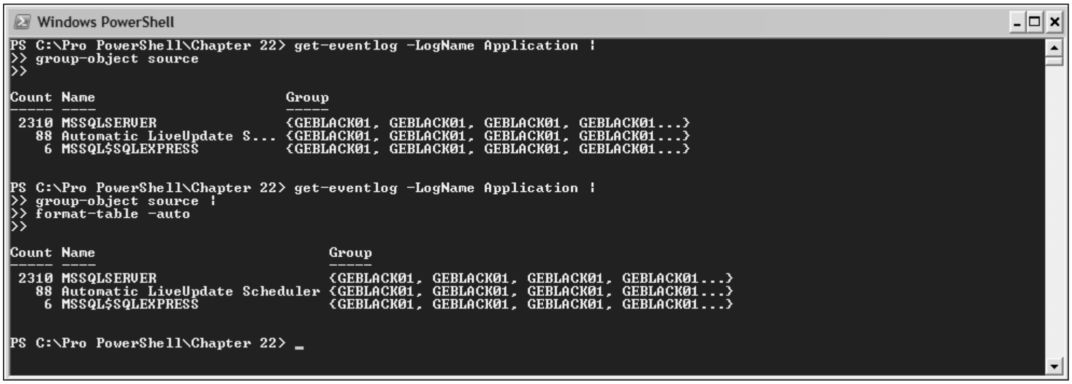


Figure 22-9

When events are being logged from several sources, it can help to insert a step using the `sort-object` cmdlet as in the following command:

```
get-eventlog -LogName System |
group-object Source |
sort-object Count -Descending |
format-table -auto
```

As you can see in Figure 22-10, it is obvious where most events are coming from.

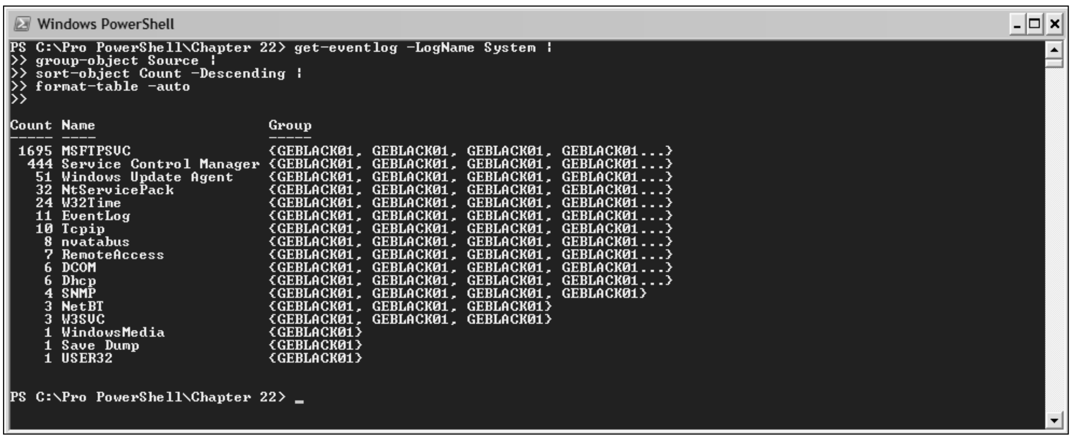


Figure 22-10

Using the preceding command, you know which source(s) are producing large numbers of logged events, but you don't have any idea of whether or not you are seeing large number of informational events logged or more serious events. Using the `where-object` cmdlet allows you to filter on the entry type.

The available entry types are listed here:

- `Information` — Indicates the successful operation of, for example, an application or service
- `Warning` — Indicates an event that may not necessarily be significant but which could indicate a current or future problem
- `Error` — Indicates a significant problem, for example the failure of a service to start
- `Success Audit` — An audited security access attempt that succeeds
- `Failure Audit` — An audited security access attempt that fails

One approach is to exclude `Information` entry types from the results. The command to do that looks like this:

```
get-eventlog -LogName System  |
where-object {$_.EntryType -ne "Information"} |
group-object Source |
sort-object Count -Descending |
format-table -auto
```

As you can see in Figure 22-11, the number of events logged that are not informational is reduced significantly compared to the total logged events in Figure 22-10. For some sources of events, such as Service Control Manager, all events appear to be informational.

The second step of the pipeline consists of this command:

```
where-object {$_.EntryType -ne "Information"}
```

The value of the `EntryType` property of each object passed along the pipeline is compared to the string literal `Information`. If the value of the `EntryType` property is not equal to the specified string literal, the object is passed to the next step of the pipeline. Otherwise it is discarded.

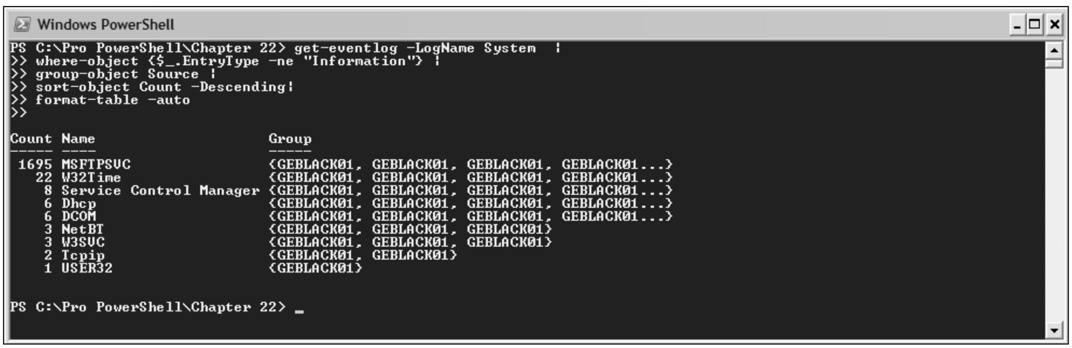


Figure 22-11

Modifying the second step of the pipeline shown in the preceding code to be

```
where-object {$_.EntryType -eq "Error"}
```

allows you to filter out all entries, except those where the entry type is `Error`. The full command then is:

```
get-eventlog -LogName System  |
where-object {$_.EntryType -eq "Error"} |
group-object Source |
sort-object Count -Descending |
format-table -auto
```

As you can see in Figure 22-12, you now have a tight handle on which sources have been producing errors on the system.

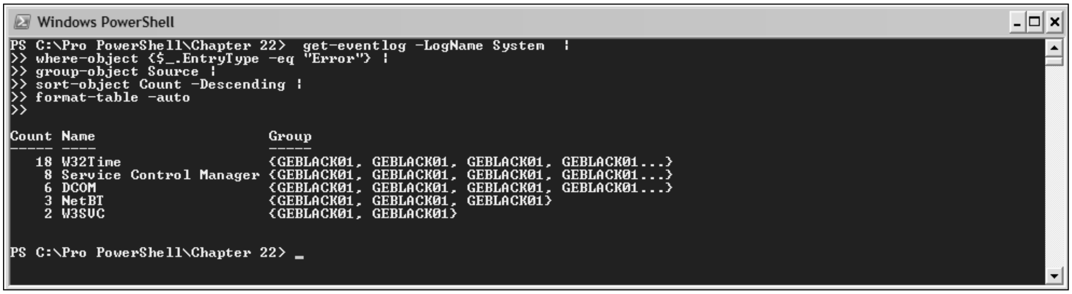


Figure 22-12

You might only be interested in recent events. You can also filtered by date and time. The `get-eventlog` cmdlet offers the `-Newest` parameter. However, that is limited in what it can do. For example, the following command displays the most recent 20 events in the Application event log:

```
get-eventlog -LogName Application -Newest 20
```

Depending on circumstances, every one of the newest 20 entries may be of the Information entry type which, almost certainly, are the events you are going to be least interested in.

However, you can create a pipeline that uses the `where-object` cmdlet to filter on entry type, then use the `select-object` cmdlet (with its `First` parameter) to display only the newest `FailureAudit` events.

The following command (which assumes you have SQL Server installed)

```
get-eventlog -LogName Application  |
where-object {$_.Source -eq "MSSQLSERVER"} |
where-object {$_.EntryType -eq "FailureAudit"}  |
select-object -first 1 |
format-list
```

uses the `where-object` cmdlet in the second step of the pipeline to discard all objects, except those where the value of the `Source` property is equal to `MSSQLSERVER`, in other words you select objects where the events have been generated by the default instance of Microsoft SQL Server. (If you don't have SQL Server installed replace appropriately in the second pipeline step.) In the third step of the pipeline, the `where-object` cmdlet is used to select only Error events. At this stage, all objects passed to the fourth step of the pipeline represent errors generated by Microsoft SQL Server. The fourth step of the pipeline uses the `select-object` cmdlet to display the first error object. The default behavior is to display the most recent error object.

Figure 22-13 shows the output from the preceding command.

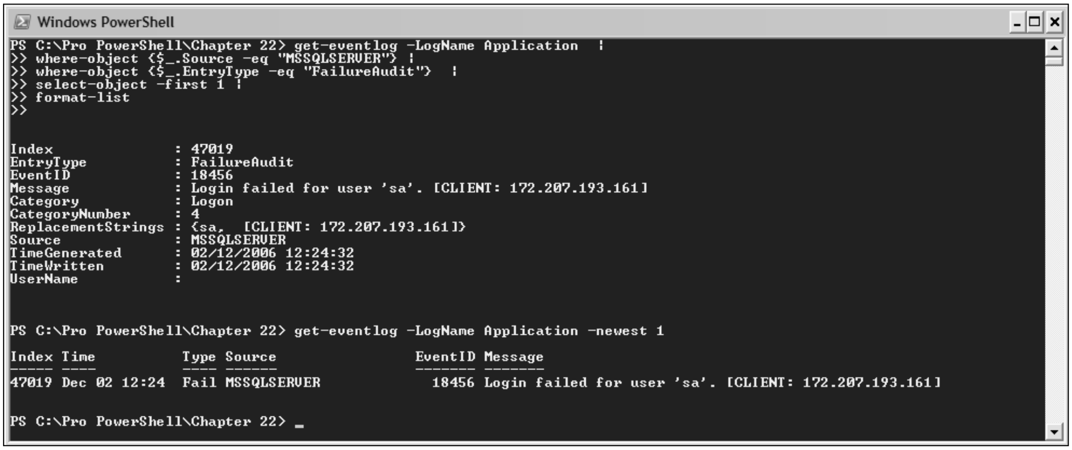


Figure 22-13

Normally, you use the `select-object` cmdlet on a sorted list. In this case, you don't need to add a `sort-object` step to the pipeline since, by default, the objects are sorted with highest Index (that is most recent) first. Notice that the Index for the event displayed in Figure 22-13 is 47019. You can confirm that the most recent error object has been selected using the following command, which displays all error events generated by the default instance of Microsoft SQL Server:

```
get-eventlog -LogName Application  |
where-object {$_.Source -eq "MSSQLSERVER"} |
where-object {$_.EntryType -eq "FailureAudit"}  |
select-object -first 15 |
format-table -auto
```

In Figure 22-14, you can confirm that the Index 47019 corresponds to the most recently generated error event from the default instance of Microsoft SQL Server. The `format-table` cmdlet is used to demonstrate unambiguously that the event displayed in Figure 22-13 is the most recent event.

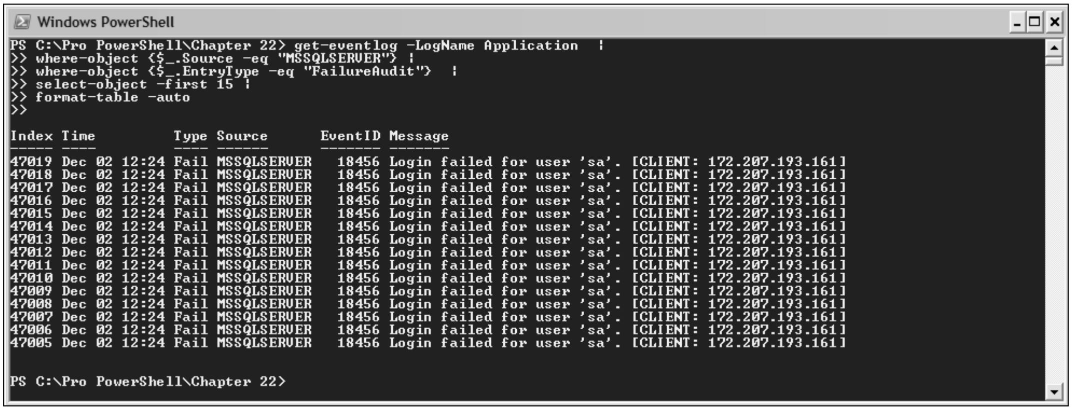


Figure 22-14

If you feel that you want a greater level of certainty that you are retrieving the most recent events, you can add another step to the pipeline, which uses the `sort-object` cmdlet to explicitly sort objects by the `TimeGenerated` property:

```
get-eventlog -LogName Application  |
where-object {$_.Source -eq "MSSQLSERVER"} |
where-object {$_.EntryType -eq "Error"}  |
sort-object TimeGenerated -descending |
select-object -first 15 |
format-table Index, TimeGenerated -auto
```

In the preceding command, the fourth step in the pipeline uses the `sort-object` cmdlet. By default, the `sort-object` cmdlet displays the lowest value first. In the context of the `TimeGenerated` property, that means that the oldest events are sorted to be first. Using the `descending` parameter causes the most recent events to be sorted to be first. Therefore, when the objects are passed to the `select-object` cmdlet in the penultimate pipeline step, it is the most recent 15 events that are selected for display. Figure 22-15 shows the results of running the command.

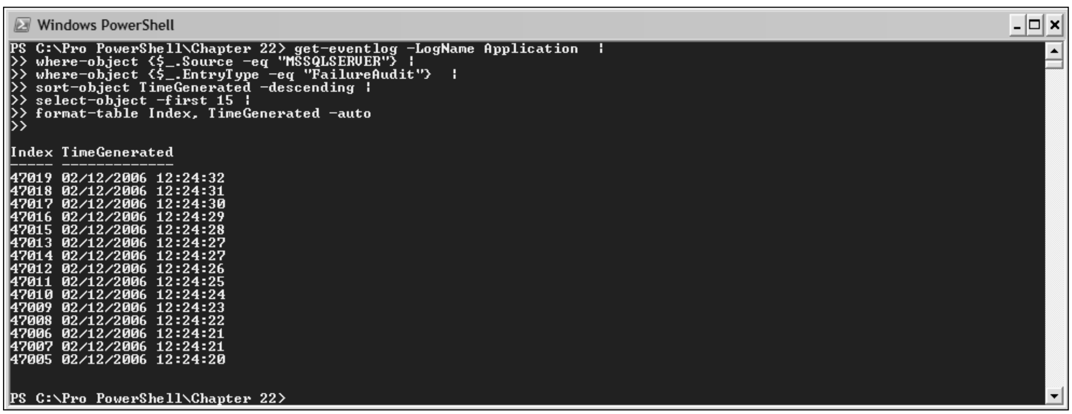


Figure 22-15

To demonstrate filtering by time generated, I will use Information events, since those are conveniently spaced on my test machine. The following command shows Information events relating to SQL Server (adapt this for another application if you don't have SQL Server installed):

```
get-eventlog Application |
where-object {$_.Source -eq "MSSQLSERVER"} |
where-object {$_.EntryType -eq "Information"}
```

As you can see in Figure 22-16, events were generated on several dates in November and December. The `TimeGenerated` property can also be used to filter events by date. In the following command, an additional pipeline step has been added to display only events generated after November 27th.

```
get-eventlog -LogName Application  |
where-object {$_.Source -eq "MSSQLSERVER"} |
where-object {$_.EntryType -eq "Information"}  |
where-object {$_.TimeGenerated -gt "2006/11/27"} |
sort-object TimeGenerated -descending |
select-object -first 15 |
format-table -auto
```

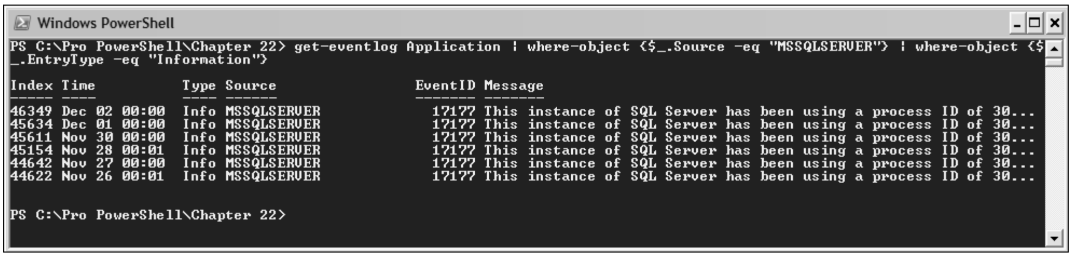


Figure 22-16

The new step uses the `where-object` cmdlet to filter by the value of the `TimeGenerated` property. As you can see in Figure 22-17, only events that occurred after 00:00:01 on November 27th that were displayed in Figure 22-16 are now displayed.

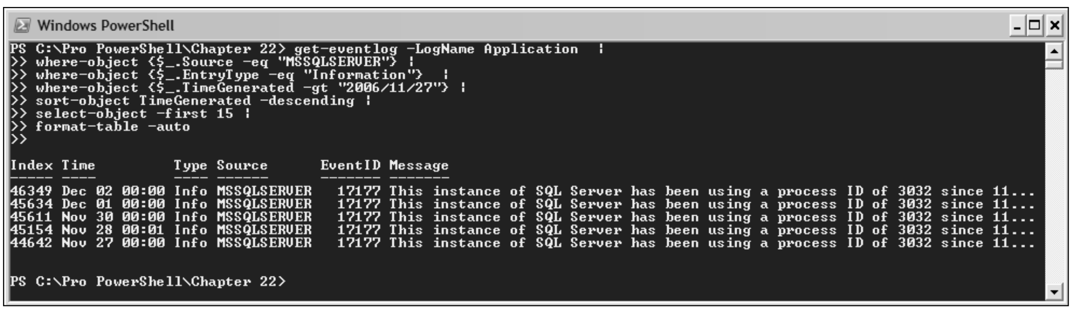


Figure 22-17

You can also combine conditions when filtering using the `where-object` cmdlet. For example, the following code displays events occurring between November 25th and November 30th. Notice that each condition is contained in parentheses. The conditions are combined using the `and` parameter of the `where-object` cmdlet.

```
get-eventlog -LogName Application  |
where-object {$_.Source -eq "MSSQLSERVER"} |
where-object {$_.EntryType -eq "Information" } |
where-object {($_.TimeGenerated -gt  "2006/11/25") -and ($_.TimeGenerated -lt
"2006/11/30")} |
sort-object TimeGenerated -descending |
format-table -auto
```

When using the and parameter with the `where-object` *cmdlet, be sure to remember to include the hyphen before it.*

Figure 22-18 shows the results of running the preceding code.

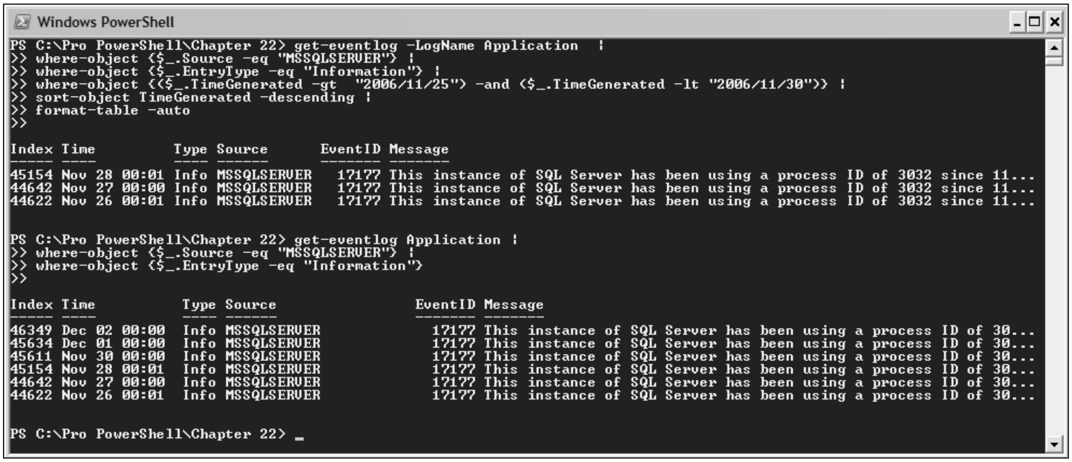


Figure 22-18

It is also possible to filter to, for example, display events which have occurred in a preceding time period. For example, to display events occurring in the last five days, you could use this command:

```
get-eventlog -LogName Application  |
where-object {$_.Source -eq "MSSQLSERVER"} |
where-object {$_.EntryType -ne "Information"} |
where-object {$_.TimeGenerated -gt (get-date).AddHours(-120)} |
sort-object TimeGenerated -descending |
format-table Index, TimeGenerated
```

I added the third step in the pipeline to reduce the number of events to be displayed. The fourth step in the pipeline uses the `get-date` cmdlet to retrieve the current date and time. Then the `AddHours()` method is used to calculate the time five days before. Since the operator used is the `gt` operator, objects are selected that represent events that occurred since 120 hours ago, that is they occurred in the last five days.

Figure 22-19 shows part of the results returned by the preceding command.

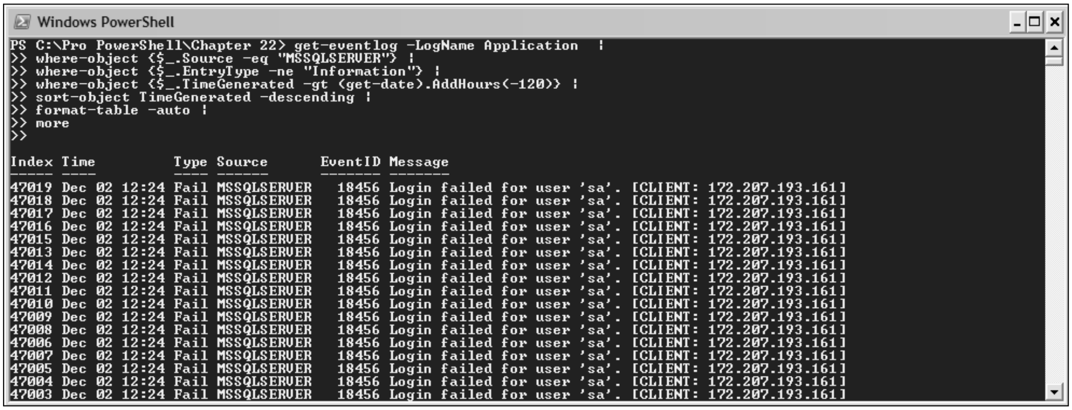


Figure 22-19

The commands

```
get-date
```

and

```
(get-date).AddHours(-120)
```

retrieve the date and time that were current when I wrote this section and the date and time five days previously. Similarly, you can use the `AddDays()` method, as in the following example command:

```
(get-date).AddDays(-10)
```

Figure 22-20 shows the relevant values at the time I was completing the writing of this chapter.

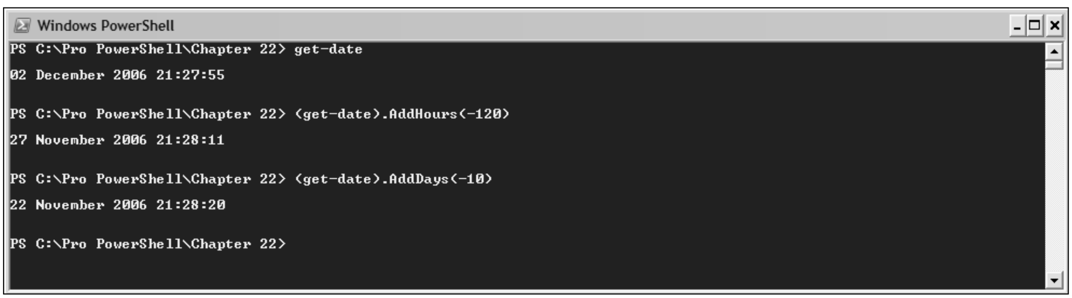


Figure 22-20

The examples so far in this chapter have shown you how to display selected events onscreen. However, you can also store textual information in a file. For example, the following command stores part of the information about the 20 most recent events on the system I am using to write this chapter:

```
get-eventlog Application -Newest 20 |
out-file "C:\Pro PowerShell\Chapter 22\ApplNewest20.evt"
```

The first step of the pipeline uses the `Newest` parameter of the `get-eventlog` cmdlet to select only the 20 newest events in the Application event log. The second step of the pipeline uses the `out-file` cmdlet to write the information passed to it to the file `C:\Pro PowerShell\Chapter 22\ApplNewest20.evt`. Figure 22-21 shows the content of the file in Notepad. Notice, however, that the content of some columns is truncated, reflecting how it would have been displayed onscreen. The file `ApplNewest20.evt` is a text file not suitable for opening in the Event Viewer.

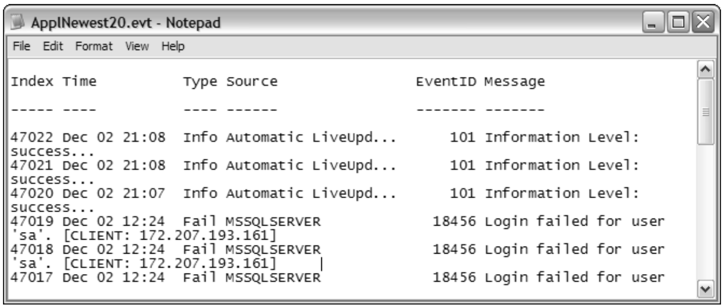


Figure 22-21

Alternatively, you could use the `get-content` cmdlet to read the file's content:

```
get-content "C:\Pro PowerShell\Chapter 22\ApplNewest20.evt"
```

In version 1.0 of Windows PowerShell, there is no cmdlet to write an event log, perhaps filtered according to specified criteria, to a file in such a way that the Event Viewer can make use of the file.

Microsoft has tools like SQL Server 2005 Profiler that are specialized for tracing what happens during execution of a particular application. Windows PowerShell allows you to create a simple event recorder for Windows PowerShell scripts. The following example script illustrates the principle of how you could use this approach.

The file `CustomLogCreator.ps1`, which illustrates the kind of thing you can do, is shown here:

```
$startTime = get-date

read-host("Start a new PowerShell instance then press the Return key.")

get-eventlog "Windows PowerShell" |
where-object {$_.TimeGenerated -gt $startTime} |
format-table |
out-file "C:\Pro PowerShell\Chapter 22\LogStartup.txt"
```

In the first line of the script, the `get-date` cmdlet is used to find the current time and assign it to the variable `$startTime`. The `read-host` cmdlet is used to pause the script. Separately, a new instance of Windows PowerShell is started manually then the Return key is pressed. That is done to raise events that occur during Windows PowerShell startup. Those events are logged in the Windows PowerShell event log.

The statement using the `get-event` log then accesses the Windows PowerShell event log. In the second step of the pipeline, the `where-object` cmdlet is used to filter events in the Windows PowerShell event log so that only events that have occurred since the date and time were assigned to the `$startTime` variable are selected. Those events are formatted as a table using the `format-table` cmdlet. In the final step of the pipeline, the `out-file` cmdlet is used to write a text representation of those events to the file `LogStartup.txt`.

Figure 22-22 shows the table of information displayed in Notepad.

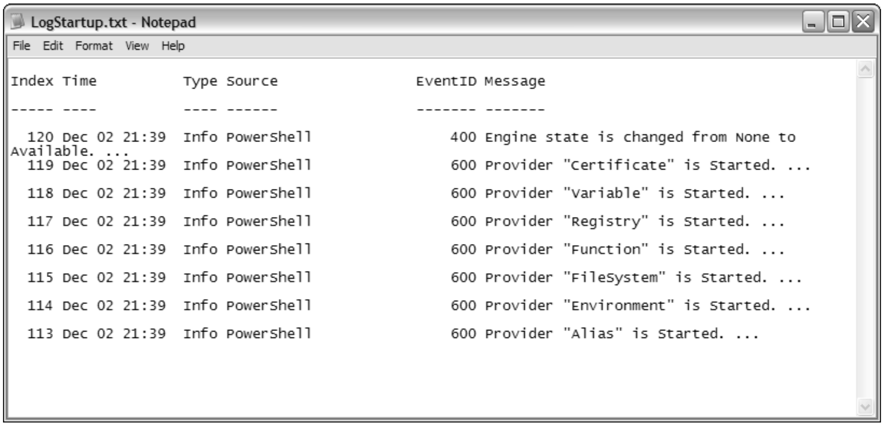


Figure 22-22

Another possibility is to incorporate some part(s) of the date and time in the filename of the custom log. The following modified example, `CustomLogCreator2.ps1`, uses the `Month` and `Day` properties of the `$startTime` variable to create a file `LogStartup122.txt`, since I ran the code on December 2nd. This approach would work if you had a folder containing multiple log files from each month.

```
$startTime = get-date

read-host("Start a new PowerShell instance then press the Return key.")

get-eventlog "Windows PowerShell" | where-object {$_.TimeGenerated -gt $startTime}
|

format-table |
out-file "C:\Pro PowerShell\Chapter
22\LogStartup$($startTime.Month)$($startTime.Day).txt"
```

Of course, you can use a greater part of the available date and time information from `$startTime` to create unique filenames.

The example illustrates the principle. In place of the `read-host` cmdlet in the preceding script, you could carry out any Windows PowerShell task or set of tasks and collect the information in a file.

Another approach would be to use the Application log and use the Windows PowerShell script to automate, say, a COM application and store the logged events during the time of interest in a separate file, which might make close scrutiny of what happens when you run the application easy.

The `eventquery.vbs` *script found in the* `C:\Windows\System32` *directory allows you to examine event logs on remote machines. A later version of Windows PowerShell is likely to provide functionality at least equivalent to that provided by* `eventquery.vbs`*.*

Summary

The `get-eventlog` cmdlet allows you to list the event logs available on the local machine and to inspect the content of a local event log of interest.

Using the `where-object`, `sort-object`, `group-object` and other cmdlets together with the properties of the `EventLogEntry` object you can filter events to focus on events of particular interest to you.

23

Working with WMI

In Windows PowerShell version 1.0, the range of available cmdlets is significant, but they cover only part of the tasks that a fully developed administrative command shell and scripting language need to cover. To fill that potential gap the PowerShell approach in version 1.0 includes support of existing technologies until such time as later versions of PowerShell provide fuller system coverage. In fact, PowerShell may continue to support existing technologies for longer than is strictly necessary, but in version 1.0 that support of legacy technologies is essential to plug the gaps that version 1.0 PowerShell cmdlets don't cover.

Another approach used alongside the Web distribution of PowerShell 1.0 is the release of specialized cmdlets designed for specific problem domains. The first such group of cmdlets is intended for use with Microsoft Exchange 2007. It is likely that other Microsoft management technologies will later have their own set of functionally related cmdlets.

One of the most important ways that PowerShell exploits existing technologies is to allow developers access to Windows Management Instrumentation, WMI, through the `get-wmiobject` cmdlet. One gap in PowerShell version 1.0 functionality is that the cmdlets in the PowerShell Web release can access resources only on the local machine. To access remote resources using PowerShell, it is initially necessary to make use of legacy technologies. Windows Management Instrumentation provides a way to achieve access to resources on remote machines.

Depending on your preferences or current knowledge (or already existing WMI scripts), you may opt to use PowerShell exclusively for what it already does in version 1.0 or progressively transition WMI scripts that use VBScript to a PowerShell context where the functionality in PowerShell version 1.0 makes that transition possible. However, if you need to access remote machines on your network, you have to use the `get-wmiobject` cmdlet in PowerShell version 1.0, since PowerShell doesn't natively support remote machine access.

Introducing Windows Management Instrumentation

Windows Management Instrumentation is a systems management technology that you can use to manage a local Windows computer or remote computers. In fact, using WMI is the only way to manage a remote Windows computer using PowerShell version 1.0.

The philosophy behind the creation of Widows Management Instrumentation was similar to the thinking applied in PowerShell. Recognition that using graphical tools can be repetitive and inefficient gave rise to a search for more efficient ways of executing tasks that have to be carried out on one machine multiple times or on multiple machines. The development of WMI took place in the broader context of the development of an approach to distributed management of computers. The Distributed Management Task Force, DMTF, drew up specifications for a cross-platform approach to distributed management of computers making use of the Web-based enterprise management, WBEM, in which Microsoft was involved.

The arrival of WMI gave administrators of Windows machines the ability to access system information and to configure and manage multiple machines. Before WMI the scripting languages that administrators used were unable to manage Windows functionality, since direct access to Windows 32 APIs was required and that wasn't generally supported using the available scripting languages.

WMI supports management of Windows 2003, Windows XP, and Windows 2000 machines. WMI and Windows Script Host enables administrators to use scripting languages such as Microsoft VBScript or ActiveState's ActivePerl, or any other scripting language that supports COM automation, to carry out administrative tasks.

WMI can, broadly, be considered as consisting of three layers (from the bottom up):

- **Managed resources** — Applications, devices, systems
- **WMI infrastructure** — WMI Scripting Library, CIM Object Manager, WMI Provider, Common Information Model
- **WMI consumers** — WMI scripts and graphical applications that use WMI under the covers

An application can be both a WMI provider and a WMI consumer. Examples include Application Center 2000 and Systems Management Server.

In the following sections, I will briefly describe each of these layers.

Managed Resources

A *managed resource* is any physical or logical system component that is exposed to Windows Management Instrumentation and can be managed by it. The range of managed resources is extensive. The following table lists a number of the resources that you can use WMI to manage.

The term "managed" in the WMI term "managed resource" does not have the connotation that the term "managed" has when applied to "managed code," that is, code created in a .NET language.

Type of resource/information	WMI class
Windows services	`Win32_Service`
Window processes	`Win32_Process`
CPU	`Win32_Processor`
Date and Time	`Win32_Date`

WMI Infrastructure

The WMI infrastructure, visualized as the middle of the three layers mentioned earlier, consists primarily of the Common Information Model Object Manager, CIMOM, the Common Information Model repository and WMI providers. In addition the middle layer includes the WMI Script Library.

WMI Script Library

The WMI Script Library contains several automation objects. Scripting languages such as VBScript and JScript use those automation objects to access the WMI functionality. These automation objects provide a consistent scripting model, making scripting WMI an easier task than working directly with the Windows APIs.

The Scripting Library makes it straightforward to write simple WMI scripts. The following script, `GetServices.vbs`, displays the name of services on the local machine.

```
strComputer = "."

Set wbemServices = GetObject("winmgmts:\\" & strComputer)
Set wbemObjectSet = wbemServices.InstancesOf("Win32_Service")

For Each wbemObject In wbemObjectSet
    WScript.Echo "Service Name: " & wbemObject.Name
Next
```

The second line of the script assigns the WMI service to the `wbemServices` variable. The value supplied to the `InstancesOf()` method, in this case `Win32_Service`, specifies what objects are of interest. By specifying another WMI class for the argument to the `InstancesOf()` method, you can explore other parts of the Windows system. The `For` loop displays a simple label with the name of each object in the `wbemObjectSet` variable.

To run the script, use this command:

```
cscript GetServices.vbs
```

Figure 23-1 shows part of the results returned.

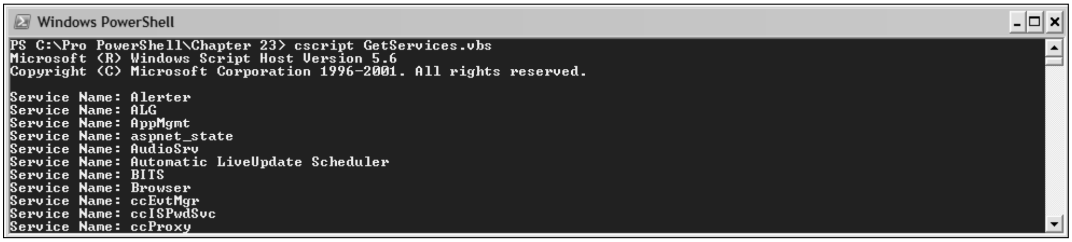


Figure 23-1

By specifying other properties of the relevant WMI object, you can display additional or alternative information onscreen.

In addition to providing objects that you will use often in VBScript scripts, the WMI Script Library includes a type library that allows you to use WMI constants in your scripts.

CIM Object Manager

The Common Information Model Object Manager controls the interaction between WMI providers and consumers. All WMI requests from WMI consumers are processed through the CIMOM. On Windows XP and Windows Server 2003, the CIMOM is implemented through `winmgmt.exe`, which is run under the service host, `svchost.exe`.

In addition to its coordinating role CIMOM also provides the following support to WMI:

- **Event processing** — Enables a WMI consumer to subscribe to selected events on a WMI managed resource.
- **Provider registration** — Registers WMI providers' location and functionality.
- **Query Processing** — Supports a WMI consumer querying a WMI managed resource. The query is in WMI Query Language, WQL.
- **Request Routing** — The CIMOM sends requests to the appropriate registered WMI provider.
- **Remote access** — WMI consumers connect to the CIMOM on a remote system to access WMI managed resources.
- **Security** — The CIMOM provides access control for WMI managed resources.

The CIM Repository

The CIM Repository contains the schema that represents configuration and management information. Each *class* in the CIM repository represents WMI managed resources. The schema in the CIM repository is based on the Common Information Model developed by the Distributed Management Task Force.

Classes in the CIM repository are grouped in namespaces. The `root\cimv2` namespace contains many classes associated with a computer and its operating system. The `Win32_Service` class that I used in the example earlier in this chapter is in the `root\cimv2` namespace.

WMI Providers

WMI providers communicate with WMI managed resources and CIMOM. Providers request information from managed resources and send information from WMI consumers to managed resources. WMI providers conceal from WMI consumers details of the Win32 APIs.

Information on WMI providers can be found online at `http://msdn2.microsoft.com/en-us/library/aa394570.aspx`*. It is also included in the documentation for the WMI Toolkit.*

WMI Consumers

WMI consumers are WMI VBScript files or Web-based or Windows-based GUI management applications that use WMI under the covers. In the context of this chapter, PowerShell can be considered a WMI consumer, since the `get-wmiobject` cmdlet uses WMI objects to carry out tasks.

WMI Tools

If you are unfamiliar with WMI, getting a grip on the many objects that form part of the Common Information Model can be a daunting task. Microsoft makes some WMI tools available that can help you explore WMI.

The WMI tools can be downloaded from `www.microsoft.com/downloads/details.aspx?FamilyID=6430f853-1120-48db-8cc5-f2abdc3ed314&DisplayLang=en`.

The CIM Studio allows you to explore a specified namespace. Figure 23-2 shows the `root\cimv2` namespace opened in CIM Studio. In the right pane, you can readily study the properties of a selected class.

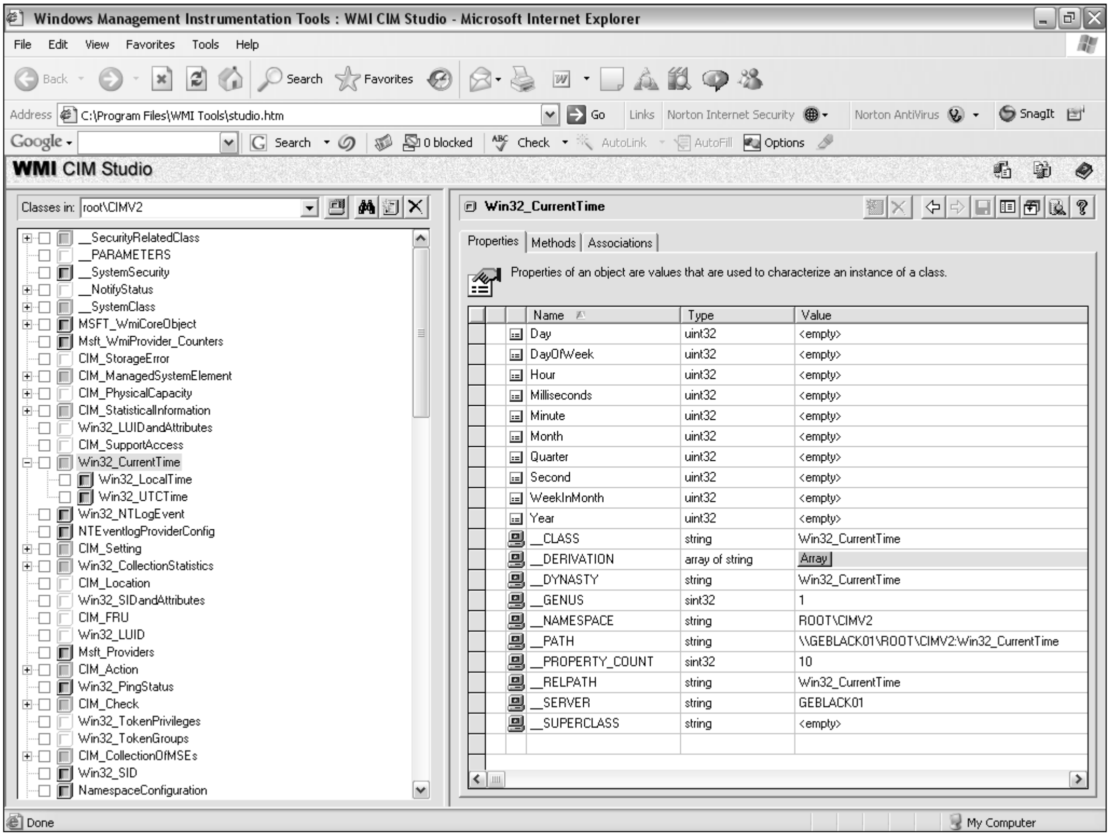


Figure 23-2

Notice in the left pane that the `Win32_CurrentTime` class is selected. In the right pane you can see the properties for that class.

In order to use the CIM Studio and CIM Object Browser in Internet Explorer, you may have to modify security settings or specifically allow active content each time you use these utilities.

The WMI Object Browser, shown in Figure 23-3, allows you to explore the values for the properties of a class in a specified namespace as they apply to a specific machine.

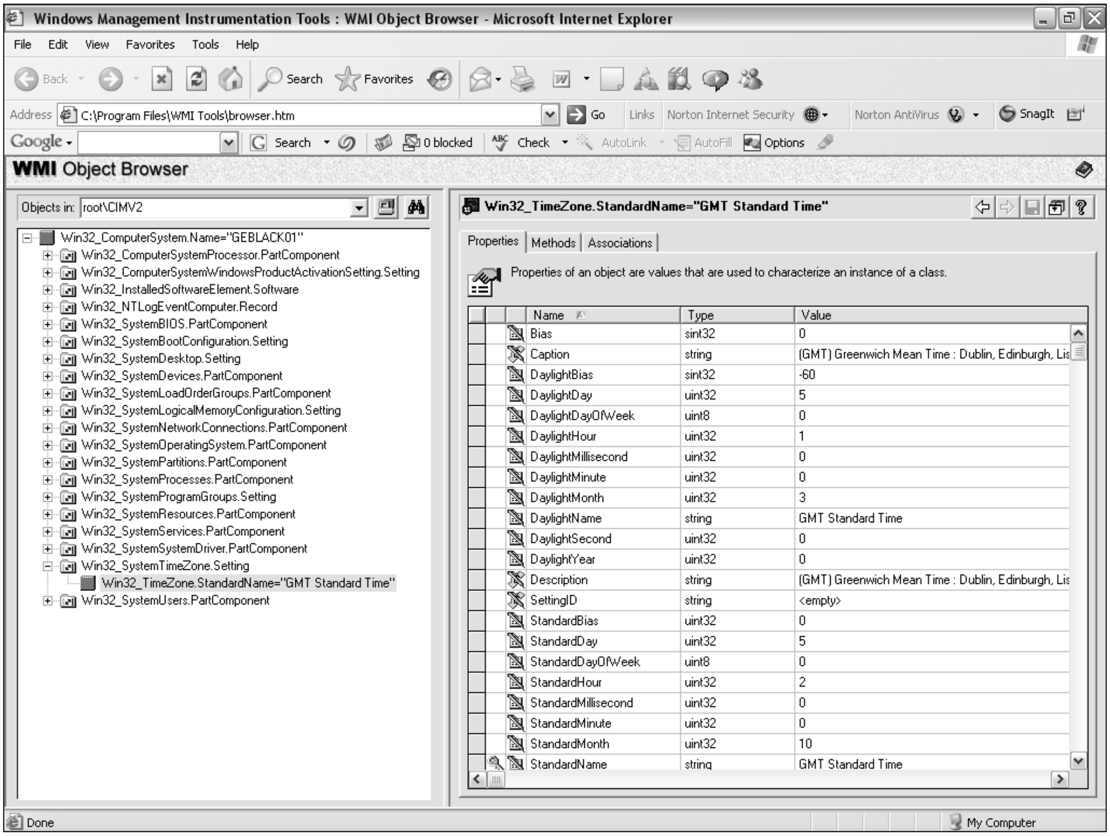


Figure 23-3

The Associations tab in the WMI Object Browser, shown in Figure 23-4, can allow you to explore visually how objects are related. Sometimes this can be very helpful in understanding object hierarchies. At other times, the visual display needs a little more thought.

The CIM Studio and CIM Object Explorer can be handy tools to explore WMI. In time you will find that using PowerShell is also an efficient way to explore these classes, even if it lacks the nice graphical output of the native WMI tools.

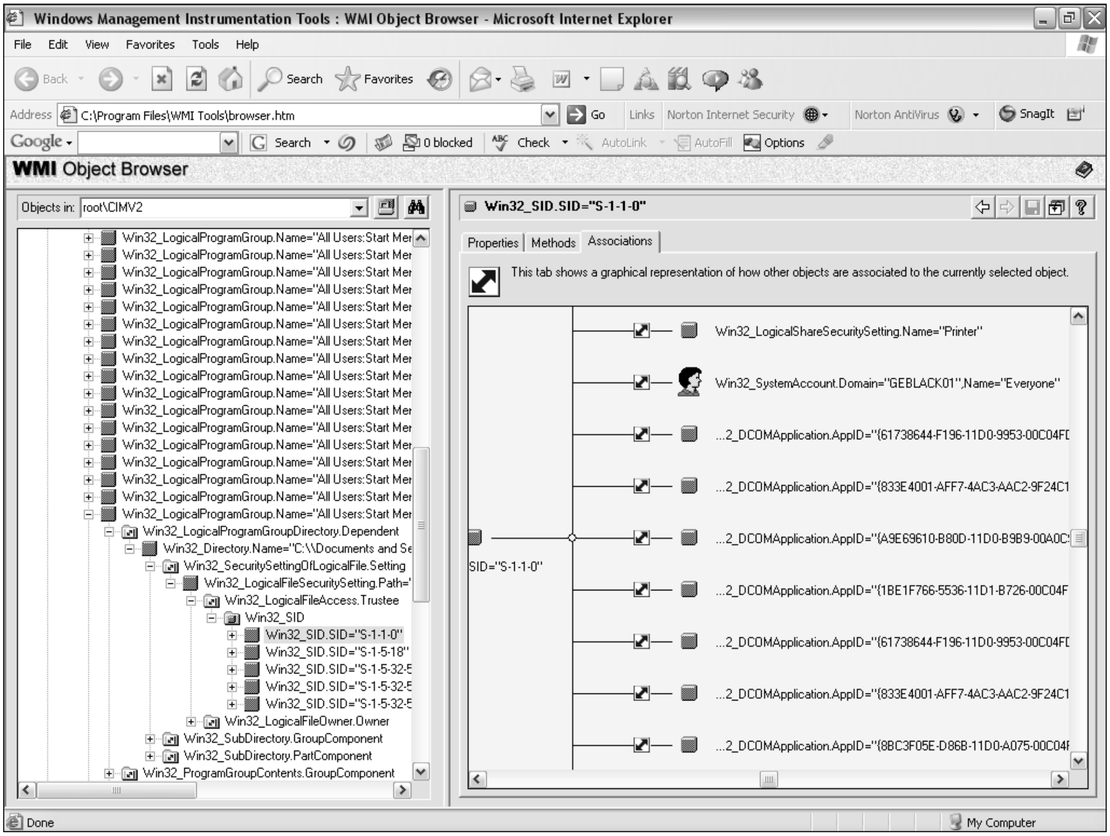


Figure 23-4

Using the get-wmiobject Cmdlet

The `get-wmiobject` cmdlet is the single cmdlet in PowerShell version 1.0 to allow you to retrieve WMI-based information. There is no comparable `set-wmiobject` cmdlet, at least not in PowerShell version 1.0.

> *Using WMI or the `get-wmiobject` cmdlet to access information from a local or remote computer makes the assumption that you have appropriate privileges to carry out the required tasks. WMI also needs to be installed on the target computer.*

To display the definition of the `get-wmiobject` cmdlet, use the following command:

```
(get-command get-wmiobject).definition
```

As you can see in Figure 23-5, there are essentially two overloads for the `get-wmiobject` cmdlet. If you use the `class` parameter, you are selecting a WMI class (or classes) on a specified machine and then displaying properties or manipulating those properties in some way. If you use the `list` parameter, you are exploring what classes are present in a specified namespace. Broadly, these approaches correspond to what you can do with the WMI Object Browser and CIM Studio, respectively.

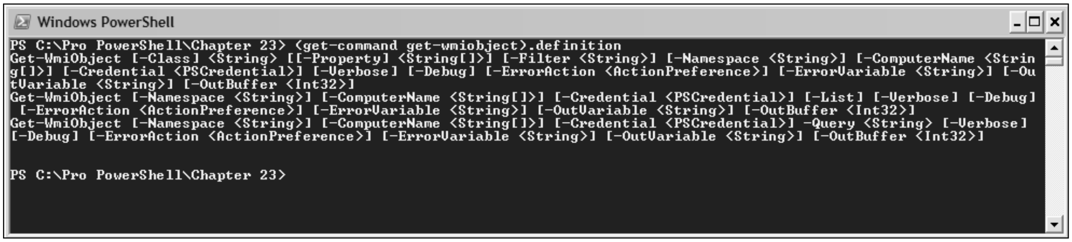


Figure 23-5

In addition to the common parameters, the `get-wmiobject` supports the following parameters.

- `Class` — Specifies a WMI class whose properties are of interest. This is a required parameter. It is a positional parameter in position 1. Wildcards are not allowed. If the List parameter is specified, the `Class` parameter is not permitted.
- `Property` — Specifies the property or properties of interest from the WMI class specified using the `Class` parameter. It is a positional parameter in position 2. The default value, if no value is explicitly supplied, is the wildcard *, which matches all properties of the WMI class.
- `Namespace` — Specifies the WMI namespace of interest. An optional, named parameter. The default value of this parameter is `root\cimv2`.
- `ComputerName` — Specifies the computer or computers where the command is to be run. This is an optional, named parameter. The default value, if no value is supplied, is `localhost`.
- `Filter` — Specifies filter elements as supported by providers. This is an optional, named parameter with no default value.
- `Credential` — Specifies a credential to be used. If a `PSCredential` object is supplied from an earlier pipeline step, it is used as is. If a user name is supplied as the value of the `Credential` parameter, the user is prompted for a password.
- `List` — Displays a list of available WMI classes. This is an optional, named parameter. If the `List` parameter is used, the `Class` parameter must not be used.
- `Query` — Specifies a WMI Query Language (WQL) statement to run. Event queries are not supported.

To demonstrate simple use of the `get-wmiobject`, execute the following command, which retrieves information about the current date and time:

```
get-wmiobject -Class Win32_CurrentTime -ComputerName .
```

The results returned are displayed in Figure 23-6. Notice that two sets of results are displayed. The first takes account of daylight savings time settings. Compare the value of the `Hour` property in the two sets of property values.

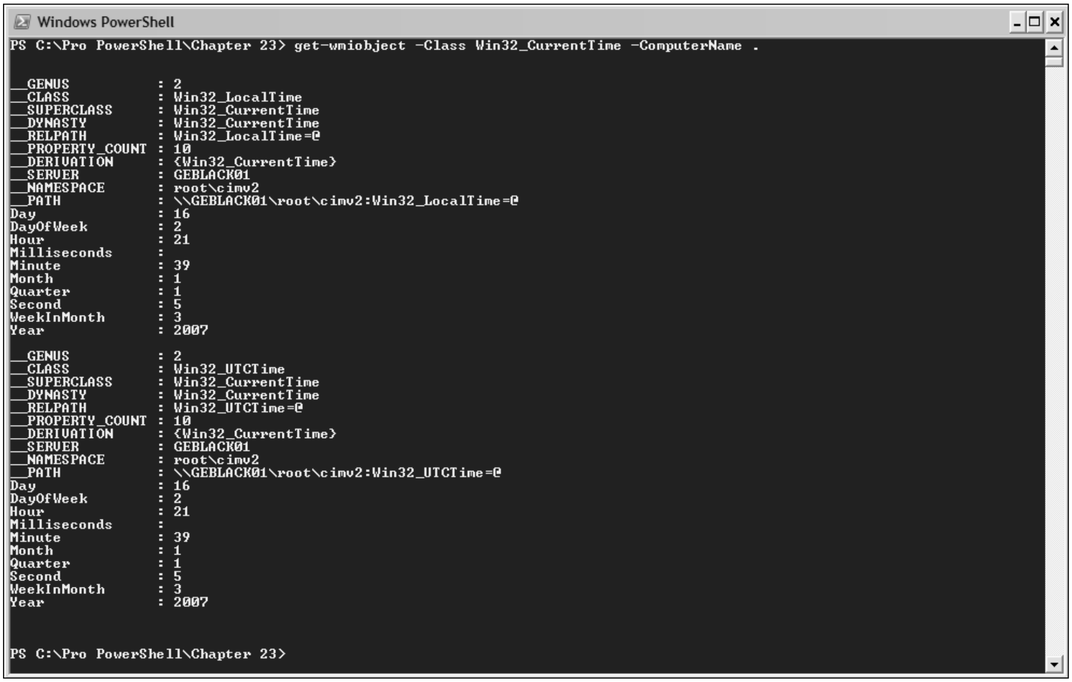


Figure 23-6

Also execute the `get-date` cmdlet, and compare the results to those shown in Figure 23-6:

```
get-date |
format-list *
```

Figure 23-7 shows the results. The key difference is that in Figure 23-6 you are seeing the properties of a WMI class. In Figure 23-7 you are seeing the properties of a .NET object.

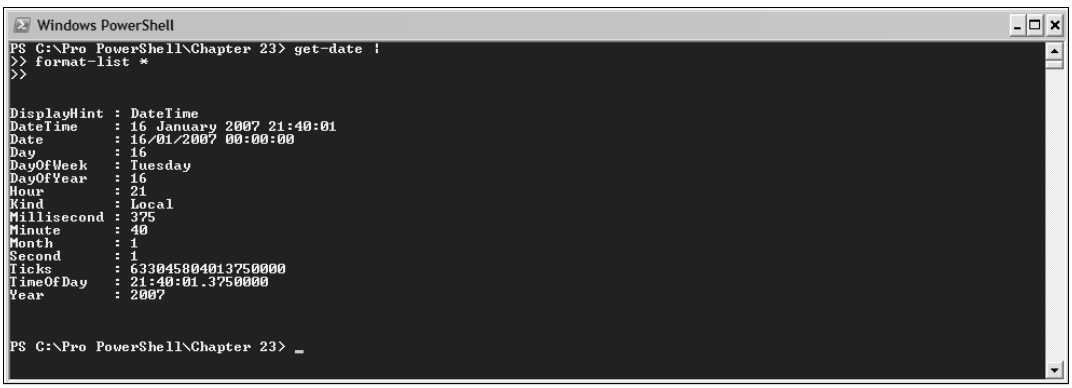


Figure 23-7

Several issues arise. A WMI object is not a .NET object. The list of available properties is different. Even where a WMI property and a .NET object property have the same name they may be of different datatypes. For example, execute the following two commands and compare the value of the `DayOfWeek` property in each result.

```
get-wmiobject -Class Win32_CurrentTime -ComputerName . |
format-table Year, DayOfWeek -auto
```

and

```
get-date |
format-table Year, DayOfWeek -auto
```

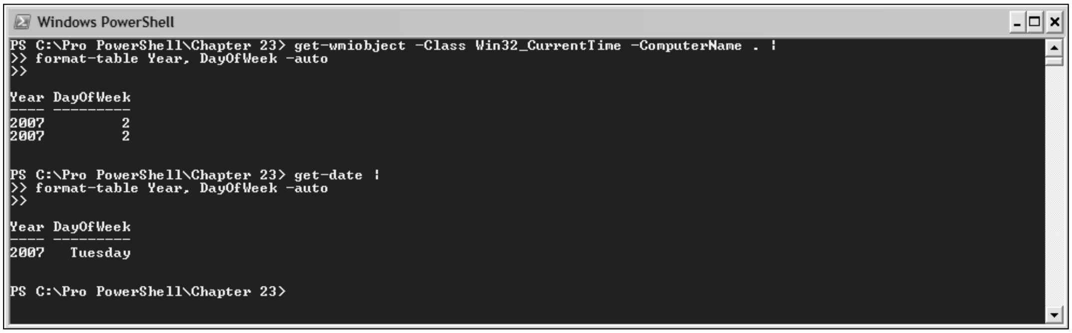


Figure 23-8

Use the following commands to display the datatypes of a relevant WMI object or .NET object:

```
get-wmiobject -Class Win32_CurrentTime -ComputerName . | get-member
get-date | get-member
```

The `DayofWeek` property for the WMI class is a `SystemUInt32`. The `DayOfWeek` property of the object produced by the `get-date` cmdlet is a `System.DayOfWeek`.

One of the advantages of PowerShell becomes obvious pretty quickly if you type in the following code, contained in the file `GetDate.vbs`:

```
strComputer = "."

Set wbemServices = GetObject("winmgmts:\\" & strComputer)
Set wbemObjectSet = wbemServices.InstancesOf("Win32_CurrentTime")

For Each wbemObject In wbemObjectSet
    WScript.Echo "Day: " & wbemObject.Day & vbCrLf & _
                 "DayOfWeek: " & wbemObject.DayOfWeek & vbCrLf & _
                 "Month: " & wbemObject.Month & vbCrLf & _
                 "Year: " & wbemObject.Year
Next
```

It quickly becomes tedious to type in multiple property names and manually add carriage returns and continuation characters. You run the command by typing

```
cscript GetDate.vbs
```

assuming that the `GetDate.vbs` file is in the current directory:

The equivalent PowerShell command

```
get-wmiobject -Class Win32_CurrentTime -ComputerName . | format-list Day,
DayOfWeek, Month, Year
```

is much easier to type. Even when you are using WMI classes the PowerShell formatting cmdlets can provide ease of use. And, of course, you can test and refine the PowerShell commands on the command line then later, if appropriate, incorporate them into a script file.

In the following sections, I will illustrate some ways in which you can use the `get-wmiobject` cmdlet. Since there are literally hundreds of WMI classes, I can only give you a hint of the range of things you can do to exploit the power of the `get-wmiobject` cmdlet.

Finding WMI Classes and Members

You can find a list of the WMI classes on the local machine by using the following command:

```
get-wmiobject -List -ComputerName .
```

If you run the command in that simple form, you will see multiple screens of information scroll past your eyes. If you use the `where-object` cmdlet, you can get some control over what is displayed. For example, you can filter the results based on WMI namespace. The following command filters classes from the `root\cimv2` namespace. It also uses the `more` alias to display one screen of results at a time.

```
get-wmiobject -List -ComputerName . |
where-object {$_.__Namespace -eq "root\cimv2"} |
more
```

Figure 23-9 shows the result of executing the preceding command.

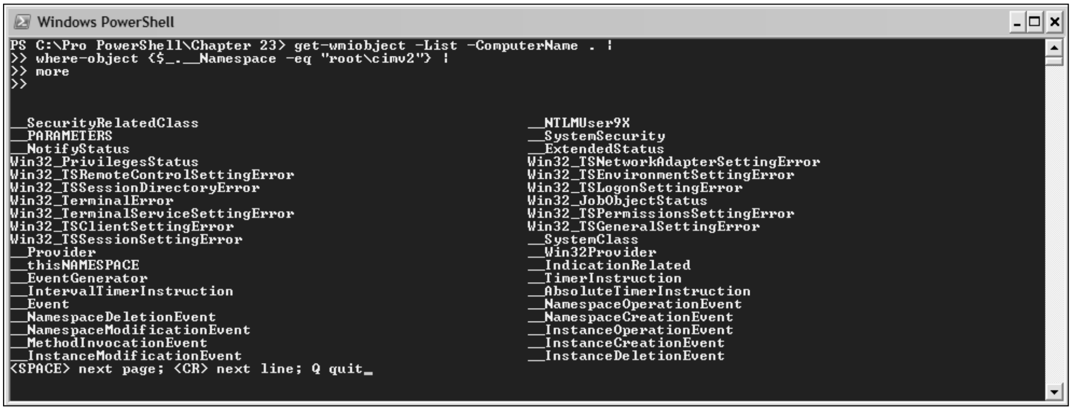


Figure 23-9

Adding another pipeline step using the `where-object` cmdlet to filter on objects that contain the character sequence Win32, then an underscore character gives us WMI classes likely to be of immediate relevance. Using the `sort-object` cmdlet sorts the results by the name of the WMI class. The modified command looks like this. I have split it across multiple lines to help you see the pipeline steps clearly.

```
get-wmiobject -List -ComputerName . |
where-object {$_.__Namespace -eq "root\cimv2"} |
where-object {$_.Name -match "Win32_.*"} |
sort-object Name |
format-table Name |
more
```

Figure 23-10 shows the results of executing the preceding command.

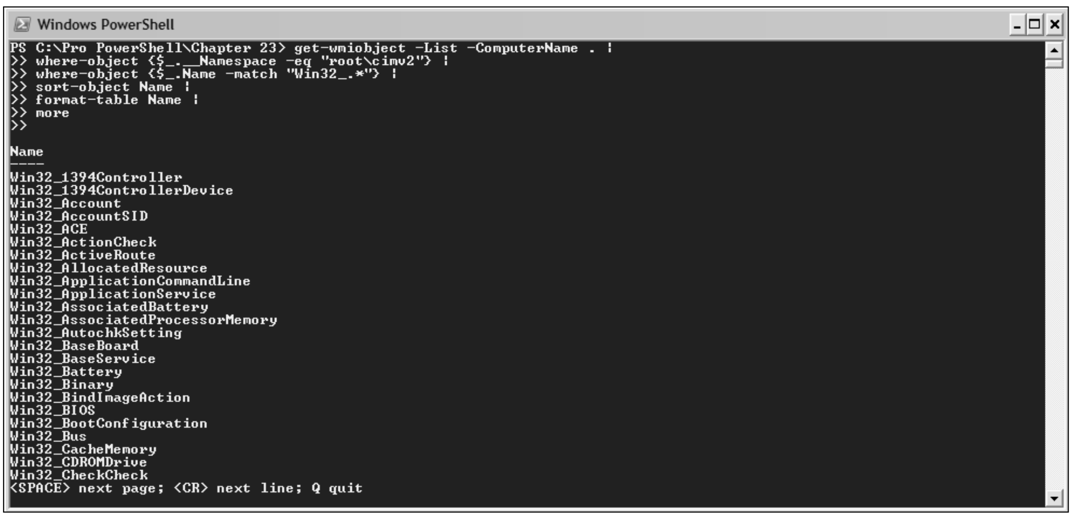


Figure 23-10

A similar set of commands can reveal the total number of WMI classes supported in PowerShell version 1.0. Execute each of these commands in turn:

```
(get-wmiobject -List).count

(get-wmiobject -List |
 where-object {$_.__Namespace -eq "root\cimv2"} ).count

(get-wmiobject -List |
where-object {$_.__Namespace -eq "root\cimv2"} |
where-object {$_.Name -match "^Win32_.*"} ).count
```

The first command uses the count property to determine the number of WMI classes that PowerShell supports in the `root\cimv2` namespace. As you can see in Figure 23-11, it is 1019 in the build I was using when writing this chapter. The second command then counts the number of WMI classes in the `root\cimv2` namespace. The number returned is the same, since no `Namespace` parameter was specified in the command and the default value for the `Namespace` parameter is `root\cimv2`. The third command filters the classes in the `root\cimv2` namespace so that only those whose `Name` property begins with `Win32_` are counted. The regular expression pattern used with the `where-object` cmdlet uses the ^ metacharacter to specify that Win32_ matches at the beginning of the class's name. The pattern `.*` matches zero or more characters. Taken together, the pattern `^Win32_.*` means "starting at the beginning of the value attempt to match `Win32_` literally then match zero or more other characters." More simply, "match names that begin with `Win32_`".

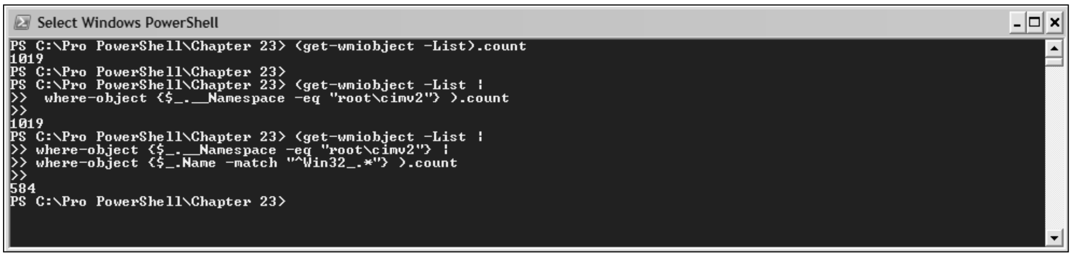


Figure 23-11

Often the name of a WMI class is fairly informative. For example, it is no surprise that the `Win32_Process` class retrieves information about processes running on a machine. Similarly, the `Win32_BIOS` class displays information about the BIOS. Since there are over 1,000 classes in the `root\cimv2` namespace alone, I won't attempt to cover available classes in any depth. The PowerShell commands and the WMI tools already described allow you to explore the WMI classes in depth, if you want to.

> *On the build that I am using some uses of the* `get-wmiobject` *intermittently hang the PowerShell window that they are executed in. At the time of writing, the cause of these hangs is unclear. Closing the PowerShell window and starting a new instance sometimes resolves the issue. Another approach is to stop and restart the WMI service.*

Once you identify a WMI class, you will typically want to work with a selected subset of its properties. If you're not using WMI routinely, you will need to be able to discover the members for the WMI objects that the `get-wmiobject` cmdlet gives you access to. The `get-member` cmdlet allows you to do that. The following command finds and displays the members of the `Win32_Process` class and pages the results onscreen.

```
get-wmiobject Win32_Process |
get-member | more
```

The `memberType` parameter allows you to focus only on, for example, properties or methods. The following command filters to display only the methods of the `Win32_Process` class and display them onscreen.

```
get-wmiobject Win32_Process |
get-member -memberType method |
more
```

Figure 23-12 shows the first screen of results.

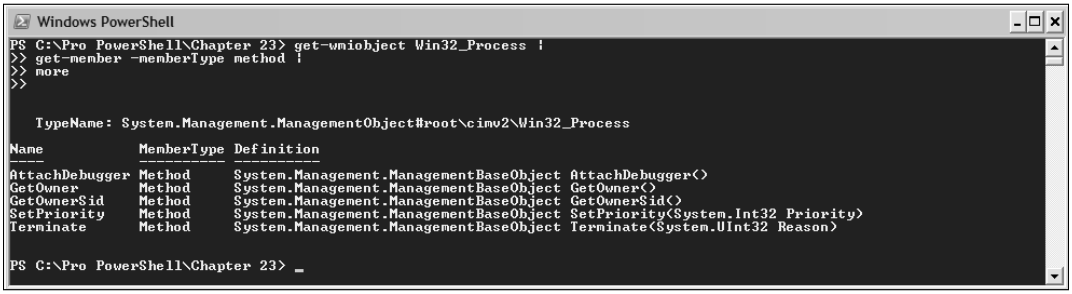


Figure 23-12

To display the properties of the Win32_Process class, simply modify the command as follows.

```
get-wmiobject Win32_Process |
get-member -memberType property | more
```

Exploring a Windows System

Now that you have seen the basics of how the `get-wmiobject` cmdlet can be used and know how to explore the `root/cimv2` namespace, I will illustrate some of the ways in which you can use the `get-wmiobject` cmdlet to explore the characteristics of a Windows system.

Characterizing the CPU

To characterize the CPU on a system, you can use the `Win32_Processor` class in the `root\cimv2` namespace. The following command retrieves information about the processor on the local machine:

```
get-wmiobject Win32_Processor |
format-list Name, MaxClockSpeed, AddressWidth, Description
```

The first step of the pipeline uses the `Win32_Processor` class to find all processors on the local machine. The `ComputerName` parameter is not expressed. However, its default value of `localhost` is assumed by the PowerShell processor. The second step displays the values of the `Name`, `MaxClockSpeed`, `AddressWidth`, and `Description` properties of the object(s) passed to it by the first pipeline step.

Figure 23-13 shows the results on a machine with a dual-core Athlon processor. Notice that the properties of each processor are reported separately. Notice, too, the disparity between the numerics displayed in the CPU's name and the value of the `MaxClockSpeed` property.

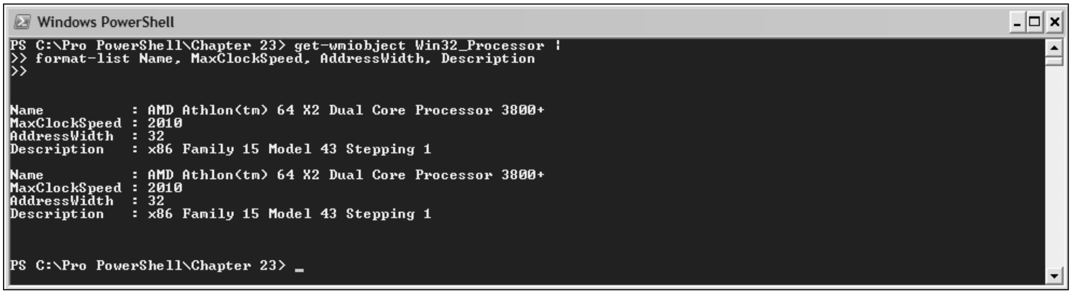


Figure 23-13

Finding Memory

One of the advantages of using PowerShell with WMI is that you can write scripts significantly more succinctly using PowerShell. For example, if you used WMI and VBScript to find out how much RAM has been installed on the local machine, you might write a script like `GetLocalMemory.vbs`:

```
strComputer = "."

Set wbemServices = GetObject("winmgmts:\\" & strComputer)
Set wbemObjectSet =

wbemServices.InstancesOf("Win32_LogicalMemoryConfiguration")

For Each wbemObject In wbemObjectSet
    WScript.Echo "Total Physical Memory (kb): " &

wbemObject.TotalPhysicalMemory
Next
```

To run that script using the `cscript` utility, you would use a command like the following:

```
cscript "C:\Pro PowerShell\Chapter 23\GetLocalMemory.vbs"
```

Of course, you could have typed in each line of the script at the command prompt. However, the PowerShell script to do the same can be written much more succinctly:

```
get-wmiobject Win32_LogicalMemoryConfiguration -ComputerName .
```

Figure 23-14 shows the execution of the two approaches.

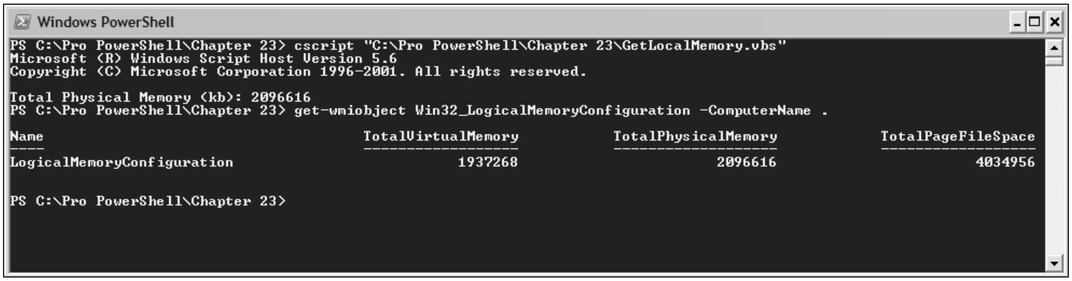


Figure 23-14

In addition to exploring the logical memory, you can also explore physical memory by using the `Win32_PhysicalMemory` class. The following command returns selected information on the banks of physical memory on the local machine.

```
get-wmiobject Win32_PhysicalMemory |
format-table BankLabel, Capacity, Tag, PositionInRow, DataWidth
```

Notice in Figure 23-15 that there are two banks of memory each of capacity 1GB on the machine being examined.

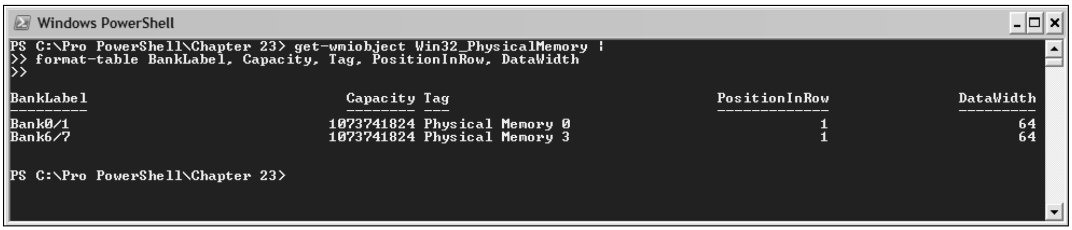


Figure 23-15

As well as exploring RAM, you can also find out information about the cache on the CPU. To do that, you use the `Win32_CacheMemory` class. The following command finds the cache memory for each processor on the local machine and displays selected properties:

```
get-wmiobject Win32_CacheMemory |
format-table DeviceID, CacheType, Purpose, InstalledSize
```

Figure 23-16 shows the results of executing the preceding command on a machine with a dual-core Athlon processor.

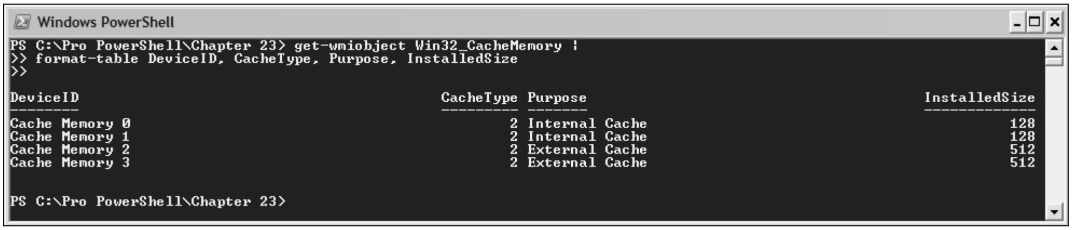


Figure 23-16

Exploring Services

To use the `get-wmiobject` cmdlet to explore services, you specify `Win32_Service` class as the value of the `Class` parameter. Using the `get-wmiobject`, you can explore services on a remote machine.

The following command returns selected information about all the services on the local machine:

```
get-wmiobject Win32_Service |
format-table Name, ProcessID, State
```

To selectively display information about running services that relate to SQL Server, you can use the following command:

```
get-wmiobject Win32_Service -ComputerName . |
where-object {$_.State -eq "Running"} |
where-object {$_.Name -match ".*SQL.*"} |
sort-object StartMode |
format-table Name, ProcessID, State, StartMode
```

The first step of the pipeline simply returns all services on the local machine. In the second step of the pipeline, the `where-object` cmdlet is used to filter objects where the value of the `State` property is `Running`. The third step of the pipeline filters objects based on a regular expression pattern. Only objects for which the value of the `Name` property contains the character sequence SQL are passed to the fourth step of the pipeline. In the fourth step of the pipeline, objects are sorted according to the value of the `StartMode` property. The final step in the pipeline displays the name, process ID, state, and start mode of each service whose properties pass the tests in the second and third steps of the pipeline.

The output of the preceding command is shown in Figure 23-17 for a Windows XP machine with SQL Server 2005 installed.

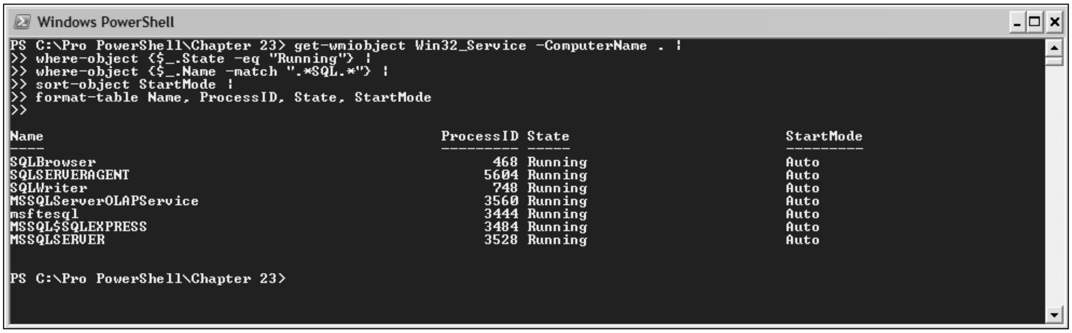


Figure 23-17

By omitting the third step in the pipeline, you can display information about all running services on the local machine.

Exploring Remote Machines

As I mentioned earlier in this chapter, WMI enables PowerShell version 1.0 to access and inspect remote machines. WMI must be installed on the remote machine you want to inspect, and you must have relevant permissions to access information on any remote machine accessed in this way.

The following commands enable you to display information about the hard drives on two remote machines. Substitute the names of machines on your network in these examples:

```
get-wmiobject -Class Win32_LogicalDisk -ComputerName Adonis |
where-object {$_.DriveType -eq 3}
```

and:

```
get-wmiobject -Class Win32_LogicalDisk -ComputerName Helios |
where-object {$_.DriveType -eq 3}
```

The second step of the pipeline specifies that the drive type is 3. That type is a hard drive.

It's not necessary to type a separate command for each machine. You can list multiple machines as the value for the `ComputerName` parameter separated by commas. The following command retrieves information about hard drives on the two machines previously accessed. The data is formatted using the `format-table` cmdlet so that it would look tidy even if you retrieved information from many machines.

```
get-wmiobject -Class Win32_LogicalDisk -ComputerName Adonis, Helios |
where-object {$_.DriveType -eq 3} |
format-table __SERVER, DeviceID, Size
```

Once you can access multiple machines, you can then use the `get-wmiobject` cmdlet for real-life admin purposes. The following example looks for machines with free space on the hard drives below a specified threshold. The initial command simply returns freespace and other properties from all specified machines:

```
get-wmiobject -Class Win32_LogicalDisk -ComputerName Adonis, Helios |
where-object {$_.DriveType -eq 3} |
format-table __SERVER, DeviceID, FreeSpace, Size
```

Adding a second step using a `where-object` cmdlet only displays information on machines where free space is less than a specified threshold. In this case, if the free space is less than 170GB, the machine on which such a drive is present is displayed.

```
get-wmiobject -Class Win32_LogicalDisk -ComputerName Adonis, Helios |
where-object {$_.DriveType -eq 3} |
where-object {$_.FreeSpace -lt 170000000000} |
format-table __SERVER, DeviceID, FreeSpace, Size
```

This command detects machines with less than 170GB free space on any hard drive.

The scope for using WMI to explore the characteristics of local and remote machines is almost limitless. I hope I have given you a flavor of what is possible. Time invested in understanding WMI will let you explore solutions specific to your own needs.

Summary

Windows PowerShell version 1.0 typically allows you to access only the local machine with limited access to shared locations on other machines on the network. To achieve greater access to networked machines, you can make use of Windows Management Instrumentation functionality from Windows PowerShell using the `get-wmiobject` cmdlet. In this chapter, I have shown you how you can make use of the `get-wmiobject` cmdlet to access the local machine and remote machines.

Index

SYMBOLS

& (ampersand), at beginning of command, 65, 68
‘ (apostrophe), 119, 416, 431–432
= (assignment operator), 72, 220–222
*** (asterisk)**
as multiplication operator, 218, 219
in `.*` pattern, 54, 80
as wildcard character, 12, 48, 49, 59, 72, 73, 125, 171–173, 200, 202
@ (at sign), in associative arrays, 249
\ (backslash), at beginning of commands, 23, 472
^ (caret)
as metacharacter, 81, 85, 290, 357, 508
in `$^` variable, 61
: (colon), 196, 346, 375, 469
, (comma), 121
{} (curly braces), for designating script blocks, 71–72
/ (division operator), 218, 219
$ (dollar sign)
at beginning of expression, 64, 65
in `$$` variable, 61
- (minus sign)
in assignment operator, 220
as subtraction operator, 218, 219
in unary operators, 226–227
`-!` **(**`-not` **operator), 225**
() (parentheses)
role in parsing, 64
using with methods, 264
% (percent sign), in assignment operator, 220, 221
. (period)
for running PowerShell scripts, 23, 472
when using PowerShell command mode, 64–66, 68
| (pipeline)
default formatter, 151–154
defined, 78
examples, 13, 14, 19–20, 41–42
grouping objects, 83–85
layout, 19
.NET objects in, 35–38, 57
overview, 19–20, 78–85
past limitations, 56–57
role in sorting objects, 81–83
separators in, 13, 78
sequence of commands, 78–81
symbol, 19, 78
syntax, 19
+ (plus sign)
as addition operator, 218, 219, 248
in assignment operators, 220
in unary operators, 226–227
? (question mark)
in `$?` variable, 61
as wildcard, 59, 72, 125, 171
“ (quotation mark, double)
at beginning of expression, 64, 65
in commands, 71, 431–432
parameters and, 119–120
‘ (quotation mark, single), 119, 416, 431–432
> (redirection operator), 18, 41, 407, 445–446
>> (redirection operator), 199, 445–446
; (semicolon), 20, 198
/ (slash)
in assignment operator, 220, 221
in division operator, 218, 219

[] (square brackets)
in PowerShell syntax for indicating .NET elements, 34, 268, 324
in syntax for designating array elements, 236
in wildcard searches, 59, 290, 429
- (subtraction operator), 218, 219
- (unary operator), 226–227
– (unary operator), 226–227
$_ **(underscore) variable, 61, 71**

A
abbreviated commands, 76–78
Access, as COM application, 303–305
Active Directory, 50
$AD **variable, 60**
add-content **cmdlet, 93**
add-history **cmdlet, 92**
addition operator (+), 218, 219, 248
add-member **cmdlet, 95**
add-psSnapin **cmdlet, 92**
administrators, setting execution policies for, 372, 373
Alias **provider, 48, 184, 185, 346**
aliases
as abbreviations for cmdlets, 77–78
background, 51–52
creating, 108–111
default, 101
displaying lists, 52–53, 107–108, 346
as drives, 48, 346–347
finding, 101–102, 347–348
list of PowerShell cmdlets for working with, 348
overview, 48–49, 77–78, 100–101
in scripts, 87
in WMI command line utility, 51–52
AllSigned **execution policy**
defined, 207
as security issue, 372, 373, 374
signed scripts and, 209
unsigned scripts and, 208
ampersand (&), at beginning of command, 65, 68
-and **operator, 225**
apostrophe ('), 119, 416, 431–432
-append **parameter, 449**
application domain, current, 353–357
Application log, 478
$Args **variable, 61**
-argumentList **parameter**
for new-object cmdlet, 311, 314, 315, 316–317
for trace-command cmdlet, 419
-arguments **parameter, 294**
arithmetic operators, 217, 218–219
arrays
adding elements, 242
associative, 249–250
concatenating, 248
creating, 235–239
defined, 235
modifying structure, 241–245
naming, 236
shortening elements, 242–243
typed, 239–240
working from end, 245–247
-asSecureString **parameter, 212, 213**
assemblies, finding, 354–355
assignment operator
equals sign (=), 72, 220–222
minus sign (-), 220
percent sign (%), 220, 221
plus sign (+), 220
slash (/), 220, 221
associative arrays, 249–250
-asString **parameter, 481**
asterisk (*)
as multiplication operator, 218, 219
in .* pattern, 54, 80
as wildcard character, 12, 48, 49, 59, 72, 73, 125, 171–173, 200, 202
at sign (@), in associative arrays, 249
automatic variables, 60–62
-autoSize **parameter, 155, 157**
-average **parameter, 201**

B
-backgroundColor **parameter, 215, 216**
backslash (\), at beginning of commands, 23, 472
-band **operator, 79**
batch files (.bat**)**
limitations, 28
maintenance, 28
versus scripting languages, 28
-begin **parameter, 261**
-bor **operator, 79**
break **statement, 394, 395**

C
C#, upgrade path to, 58
caret (^)
as metacharacter, 81, 85, 290, 357, 508
in $^ variable, 61

case
.NET Framework and, 18, 326–327
in PowerShell cmdlets, 18, 92, 327
-category **parameter, 400**
-categoryActivity **parameter, 400**
-categoryReason **parameter, 400**
-categoryTargetName **parameter, 400**
-categoryTargetType **parameter, 400**
-ccontains **operator, 80**
cd **alias, 49**
-ceq **operator, 80, 223**
Certificate **namespace, 374–376**
-certificate **parameter, 377**
Certificate **provider, 184, 185, 346**
certificates
code-signing, 376–379
creating, 376–377
as drives, 51, 374–375, 376
finding, 377–378
-cge **operator, 79, 223**
-cgt **operator, 79, 223**
-character **parameter, 201**
child items, obtaining, 55. *See also* get-childItem **cmdlet**
-childPath **parameter, 452**
CIM repository, 499
CIM Studio tool, 499–500
CIMOM (Common Information Model Object Manager), 497, 498
classes, .NET Framework
calling, 34
displaying by using get-member, 35, 36
finding information about, 324–333
library, 36, 37–38
-cle **operator, 79, 223**
clear **command, 17**
clear-content **cmdlet, 93**
clear-host **cmdlet, 17**
clear-item **cmdlet, 93**
clear-itemProperty **cmdlet, 93**
clear-variable **cmdlet, 95, 231–232**
-clike **operator, 80, 223**
Clone() **method, 257, 265**
closing Windows PowerShell, 10
cls **cmdlet, 18, 52**
-clt **operator, 79, 223**
-cmatch **operator, 80, 224**
CMD.exe
background, 27
exiting PowerShell to prompt, 10
limitations, 27–29
versus PowerShell, 428, 455
similarity of PowerShell to prompt, 17
CmdletInfo **object, 152, 153**
cmdlets
aliases for abbreviating, 77–78
availability, 11–12, 75–76
case-insensitivity, 18, 92, 327
combining in pipelines, 19–20, 78–85
completing by using Tab key, 76–77, 112–113, 443–444
cycling through recently used, 18
defined, 11
exploring, 11–14
for file actions, 449–450
finding, 11–14
finding parameters, 121–124
focusing search, 12
getting help with, 14–17
getting recent-use history, 18
grouping, 83–85
naming scheme, 57–58, 85
overview, 20–21
parameters and, 117–131
separator between, 20
syntax, 20–21
testing combinations, 21–23
verbosity issue, 85–87
verbs as, 13
viewing list, 11–12
ways to abbreviate, 76–78
for working with paths, 450–453
-cne **operator, 80, 223**
-cnotcontains **operator, 80**
-cnotlike **operator, 80, 224**
-cnotmatch **operator, 80, 224**
code debugging
handling syntax errors, 403–408
as PowerShell feature, 59
set-psDebug cmdlet, 408–413
tracing, 418–423
write-psDebug cmdlet, 413–418
colon (:), 196, 346, 375, 469
COM (Component Object Model)
accessing, 56
creating objects by using new-object cmdlet, 293–294
Internet Explorer and, 294–299
Microsoft Access and, 303–305
Microsoft Excel and, 302–303
Microsoft Word and, 301–302
network shares and, 305–306

COM (Component Object Model) (continued)
object scripting, 43–44
PowerShell support, 293
role of synthetic types, 306–308
specific applications and, 294–308
Windows Script Host and, 30, 299–301
comma (,), 121
command line, PowerShell
applying in exploratory way, 39–41
approaches to parsing, 63–69
batch file tools versus scripting languages, 28
command mode versus expression mode, 63–69
existing utilities, 53–55
tool history, 26–27
tool inconsistency, 28
using utilities from, 54
Windows utilities, 53–55
command mode
examples, 66–69
versus expression mode, 63–65
mixing expressions with, 69
`-command` **parameter, 419**
`-Command` **PowerShell startup option, 10**
command prompt
>> as, 19
creating, 113–115
customizing, 113–115
command shell providers. *See* providers
commands, PowerShell, using Tab key to complete, 76–77, 112–113, 443–444. *See also* cmdlets; scripts, PowerShell
Common Information Model Object Manager (CIMOM), 497, 498
`-comObject` **parameter, 43, 56, 294, 311**
`Compare()` **method, 265, 268–270**
`compare-object` **cmdlet, 95**
`CompareTo()` **method, 265, 271**
comparing strings, 268–271
comparison operators, 217, 222–224
completing commands by using Tab key
cmdlet examples, 443–444
as PowerShell abbreviated command form, 76–77
registry example, 112–113
when not to use, 113
Component Object Model (COM)
accessing, 56
creating objects by using `new-object` cmdlet, 293–294
Internet Explorer and, 294–299
Microsoft Access and, 303–305
Microsoft Excel and, 302–303
Microsoft Word and, 301–302
network shares and, 305–306
object scripting, 43–44
PowerShell support, 293
role of synthetic types, 306–308
specific applications and, 294–308
Windows Script Host and, 30, 299–301
concatenated strings, 65
concatenating arrays, 248
conditional expressions, 250–256
`-confirm` **parameter**
for `clear-variable` cmdlet, 231
as common parameter, 132, 165, 166
for `new-item` cmdlet, 203
for `new-psDrive` cmdlet, 204
for `new-variable` cmdlet, 229
for `remove-psDrive` cmdlet, 187
for `remove-variable` cmdlet, 233
for `set-authenticodeSignature` cmdlet, 377
for `set-executionPolicy` cmdlet, 371
for `set-variable` cmdlet, 228
for `stop-service` cmdlet, 180–181
`$ConfirmPreference` **variable, 61**
console files, loading, 90
`$ConsoleFileName` **variable, 61**
`Constant` **option, 348–349, 388**
consumers, WMI, 499
`Contains()` **method, 265, 271–272**
`-contains` **operator, 79**
`continue` **statement, 394–395**
`ConvertFrom-SecureString` **cmdlet, 94**
`convert-path` **cmdlet, 93, 450**
`ConvertTo-html` **cmdlet, 95**
`ConvertTo-SecureString` **cmdlet, 94**
copying strings, 267
`copy-item` **cmdlet, 93**
`copy-itemProperty` **cmdlet, 93**
`CopyTo()` **method, 265, 272–273**
core library, .NET, 355–358
Core snapin, 90–92
`count` **property, 21, 22**
CPUs. *See* `Win32_Processor` **WMI class**
`-credential` **parameter**
for `get-content` cmdlet, 197
for `join-path` cmdlet, 452
for `new-item` cmdlet, 203, 204
for `remove-item` cmdlet, 167
for `resolve-path` cmdlet, 453
for `test-path` cmdlet, 451

curly braces ({}), for designating script blocks, 71–72
current application domain, 353–357
current working location, system state information, 338, 340–341
`CurrentDomain` **property, 354**
custom drives, 447–448

D

data stores, as drives, 45
databases, accessing data in Access databases from PowerShell command line, 303–305
date and time examples, 22, 34–38
`DateTime` **object**
  creating by casting strings, 318–319
  creating by using `new-cmdlet` cmdlet, 312–317
  role in casting strings, 288–289
  `ToString()` method, 22
`DayOfWeek` **property, 288**
`-debug` **parameter, 132**
`-debugger` **parameter, 419**
debugging code
  handling syntax errors, 403–408
  as PowerShell feature, 59
  `set-psDebug` cmdlet, 408–413
  tracing, 418–423
  `write-psDebug` cmdlet, 413–418
`$DebugPreference` **variable, 61, 414, 415**
decimal point (.)
  for running PowerShell scripts, 23, 72
  when using PowerShell command mode, 64, 65, 66, 68
default formatter, 151–154
`default` **option, 255–256**
`Definition` **property, 323, 349**
`-delimiter` **parameter, 197, 198**
`-description` **parameter**
  for `new-psDrive` cmdlet, 204
  for `new-variable` cmdlet, 229
  for `set-variable` cmdlet, 228
developers, setting execution policies for, 372, 373
`dir` **alias, 47, 48, 52, 55, 78, 433**
`DirectoryInfo` **object, 434–435**
`-displayErrors` **parameter, 156**
`-displayName` **parameter, 178**
division operator (/), 218, 219
dollar sign ($)
  at beginning of expression, 64, 65
  in `$$` variable, 61
DOS machines, 26
dot (.) notation
  for running PowerShell scripts, 23, 72
  when using PowerShell command mode, 64, 65, 66, 68
`do/while` **statement, 259–260**
drives. *See also* `get-psDrive` **cmdlet**
  aliases as, 48, 346–347
  certificates as, 51, 346, 374–375, 376
  creating, 187, 204–205
  custom, creating, 447–448
  defined, 45
  finding, 434
  finding information about, 425–426
  in fully qualified path names, 427
  functions as, 349–350
  mapping, 305–306
  namespaces as, 45–51
  PowerShell definition, 45
  relationship to providers, 45, 186
  removing, 94, 187–188
  variables as, 49, 346, 350

E

`elseif` **clause, 252–253**
`-encoding` **parameter**
  for `get-content` cmdlet, 197
  for `out-file` cmdlet, 449
`-end` **parameter, 261**
end users, setting execution policies for, 372, 373
`EndsWith()` **method, 265, 273–274**
`Environment` **provider, 184, 185, 346, 351, 468–469**
environment variables
  defined, 465
  editing information about, 467–468
  exploring by using `get-childItem` cmdlet, 470–471
  finding information about, 466–467
  modifying, 466, 471–473
  overview, 465–468
  as PowerShell tool, 351–353
`-eq` **operator**
  as comparison operator, 223
  versus = operator, 72
  using with `where-object` cmdlet, 79, 144
equal sign (=), as assignment operator, 72, 220–222. *See also* `-eq` **operator**
`Equals()` **method, 265, 274–275**
error handling, system state information, 338, 345–346
`$Error` **variable, 61, 383–387, 388**

-errorAction parameter, as common parameter, 132, 397–398
$ErrorActionPreference variable, 61, 390–392, 397, 398
-errorId parameter, 400
-errorRecord parameter, 400
errors
nonterminating, 381, 382, 392
related variables, 345–346, 383–392
syntax, 403–423
system, 381–401
terminating, 381, 382
trapping, 392–397
-errorVariable parameter, as common parameter, 132, 399
$ErrorView variable, 61, 389–390
event logs
displaying entries in Windows Event Viewer, 478
displaying entries onscreen by using PowerShell, 480–492
overview, 477–480
PowerShell versions, 477
writing entries to file, 492–494
Event Viewer
Filter tab, 478, 479
opening, 477
viewing logs, 478
Excel, as COM application, 302–303
-exception parameter, 400
Exchange Management Shell, 50
Exchange Server 2007, 38, 39, 50
exclamation point, in -! operator, 225
-exclude parameter
for clear-variable cmdlet, 231
for get-childitem cmdlet, 191
for get-content cmdlet, 197
for get-variable cmdlet, 230
for remove-item cmdlet, 167
for remove-variable cmdlet, 232
for set-variable cmdlet, 228
for stop-service cmdlet, 178
for test-path cmdlet, 451
-excludeProperty parameter, 144
execution policies
checking by using get-executionPolicy cmdlet, 209–212
defined, 207
enabling scripts by using set-executionPolicy cmdlet, 9, 23
overview, 207–209
as security issue, 370–374
$ExecutionContext variable, 61
-executionPolicy parameter, 371, 373
exiting Windows PowerShell, 10
-expand parameter, 156
-expandProperty parameter, 144, 146
export-alias cmdlet, 95, 105–106, 348
export-clixml cmdlet, 95
export-console cmdlet, 92
export-csv cmdlet, 95
expression mode
versus command mode, 63–65
examples, 65–66
mixing commands with, 69
-expression parameter, 419
expressions
conditional, 250–256
regular, 289–291
extended wildcards, 59–60

F

$False variable, 61
file system
cmdlets for file actions, 449–450
finding file characteristics, 436–439
finding files and folders, 425, 427–431
finding hidden files, 442–443
removing items, 166–175
FileInfo object
retrieving, 434, 435
select-object cmdlet and, 440, 441
working with methods, 436–437
working with properties, 438–439
-filePath parameter
for out-file cmdlet, 449
for set-authenticodeSignature cmdlet, 377, 378
for trace-command cmdlet, 419
files and folders
finding, 434–436
finding file characteristics, 436–439
FileSystem provider
defined, 184, 185, 346, 425
multiple drives and, 195–196
overview, 45, 46, 47, 426
-filter parameter
for get-childitem cmdlet, 191
for get-content cmdlet, 197
for remove-item cmdlet, 167
for test-path cmdlet, 451

filtering processes
using `where-object` cmdlet, 41, 71–75, 79–81, 137–144
using wildcards, 72–73
filters, 349–350
`-filterScript` **parameter, 138, 142–143**
finding
aliases, 101–102, 138, 347–348
assemblies, 354–355
certificates, 377–378
cmdlets, 11–14
drives, 434
file characteristics, 436–439
files and folders, 434–436
hidden files, 442–443
information about drives, 425–426
information about environment variables, 466–467
information about .NET classes, 324–333
members of .NET Framework objects by using `get-member` cmdlet, 35, 36, 50, 320–323
parameters, 121–124
PowerShell commands, 11–14, 75–76
providers, 183–184, 425
services that are running by using `get-service` cmdlet, 74–75, 138–140
Windows processes that are running by using `get-process` cmdlet, 69–71
`findstr` **command, 53, 54**
`-first` **parameter, 145, 148–150**
`for` **statement, 256–258, 382**
`-force` **parameter**
for `clear-variable` cmdlet, 231
for `format-table` cmdlet, 156
for `get-childItem` cmdlet, 191, 442–443
for `get-content` cmdlet, 197
for `new-item` cmdlet, 203
for `new-variable` cmdlet, 229
for `remove-item` cmdlet, 166
for `remove-psDrive` cmdlet, 187
for `remove-variable` cmdlet, 232
for `set-variable` cmdlet, 228
for `stop-service` cmdlet, 178
for `trace-command` cmdlet, 419
`foreach` **statement, 238, 260–261**
`foreach-object` **cmdlet, 92**
`-foregroundColor` **parameter, 215, 216**
`format-custom` **cmdlet, 95**
`$FormatEnumerationLimit` **variable, 61**
`format-list` **cmdlet**
adding to final pipeline step, 323
defined, 95
displaying information, 57, 186
example, 58
`-force` parameter in, 386
overview, 161–162
`format-table` **cmdlet**
`-autoSize` parameter in, 157
defined, 95, 155
displaying information, 57
`-groupBy` parameter in, 158–159
`-hideTableHeaders` parameter in, 158
list of parameters, 155–156
`-property` parameter in, 156–157
specifying column labels, 159–161
specifying column widths, 159–161
`format-wide` **cmdlet, 95**
`-full` **parameter, 60**
`FullName` **property, 355**
fully qualified path names, 427–430
`Function` **drive, 349–350, 431**
`Function` **provider, 184, 185, 346**
functions, 349–350

G

`gci` **alias, 52. *See also*** `get-childitem` **cmdlet**
`-ge` **operator, 79, 144, 223**
`get` **verb, 57**
`get-acl` **cmdlet, 94**
`get-alias` **cmdlet**
versus `Alias` provider, 48
defined, 95, 348
overview, 107–108
using to find available aliases, 52
verbosity example, 85–86
`GetAssemblies()` **method, 354**
`get-authenticodeSignature` **cmdlet, 94, 376, 378**
`get_Chars()` **method, 265, 275**
`get-childItem` **cmdlet**
aliases, 48–49, 55, 78
defined, 93
versus `dir` alias, 47
examples, 41–42, 68–69
`-force` parameter in, 442–443
overview, 191–194
`Registry` provider and, 47–48
retrieving aliases, 103–104
retrieving environmental variable information, 351–353
`get_ChildNodes()` **method, 289**
`get-command` **cmdlet**
defined, 92
versus `findstr` command, 54
using to find commands, 11, 12, 13–14, 20–21, 75–76

get-content **cmdlet, 21–22, 54, 93, 196–201**
get-credential **cmdlet, 94**
get-culture **cmdlet, 95**
get_CurrentDomain() **method, 354**
get-date **cmdlet, 21, 22, 36–37, 56, 95, 490–491**
GetEnumerator() **method, 265**
get-eventlog **cmdlet, 93, 480–494**
get-executionPolicy **cmdlet, 94, 371**
GetHashCode() **method, 265**
get-help **cmdlet**
  defined, 92
  displaying detail, 60
  finding parameters, 121–124
  overview, 14–17
get-history **cmdlet, 18, 92**
get-host **cmdlet, 95**
get-item **cmdlet, 93**
get-itemProperty **cmdlet, 93**
get_Length() **method, 265, 275**
get-location **cmdlet, 93, 194–196, 338–345**
get-member **cmdlet, 35, 36, 50, 95, 320–323**
GetMember() **method, 326–328**
GetMembers() **method, 324–325**
GetMethod() **method, 329**
GetMethods() **method, 328–329**
get-pfxCertificate **cmdlet, 94**
get-process **cmdlet**
  defined, 93
  filtering results by using where-object cmdlet, 73, 81, 138–140
  finding running Windows processes, 69–71
  .NET Framework System.Diagnostics.Process objects and, 34
  parameters for, 118–121
  in scripting example, 21, 22
  sorting processes by using sort-object cmdlet, 81–83
GetProperties() **method, 330–331**
GetProperty() **method, 331–333**
get-psDrive **cmdlet**
  defined, 45, 93
  versus get-psProvider cmdlet, 102–103
  using to find available drives, 184–188
get-psProvider **cmdlet**
  defined, 45, 93
  versus get-psDrive cmdlet, 102–103
  using to find available providers, 183–184
get-psSnapin **cmdlet, 90–91, 92**
get-service **cmdlet**
  compared with Services MMC snap-in, 73–74
  defined, 57, 93, 357
  examples, 125–127
  finding running services, 74–75, 138–140
  overview, 358–359
get-traceSource **cmdlet, 95, 422–423**
GetType() **method, 265, 276**
GetTypeCode() **method, 265, 276**
get-UICulture **cmdlet, 95**
get-unique **cmdlet, 95**
get-variable **cmdlet, 95, 230–231**
get-wmiObject **cmdlet**
  defined, 93, 502
  examples, 503–509
  exploring Windows system characteristics, 509–514
  finding WMI classes and members, 506–509
  list of parameters, 503
graphical user interface (GUI), 26
greater-than operator, 79, 144, 223, 407
greater-than-or-equal-to-operator, 79, 144, 223
-groupBy **parameter**
  format-table cmdlet and, 155, 158–159
  group-object cmdlet and, 158–159
group-object **cmdlet**
  defined, 95
  examples, 152
  overview, 83–85
  as verb-noun naming scheme example, 57–58
-gt **operator, 79, 144, 223, 407**
GUI (graphical user interface), 26

H

help **alias, 111–112**
help **function, 15**
-Help **PowerShell startup option, 10–11. *See also* get-help cmdlet**
hidden files, finding, 442–443
-hideTableHeaders **parameter, 155, 158**
hiding table headers, 158
HKEY_CLASSES_ROOT **hive, 456**
HKEY_CURRENT_CONFIG **hive, 456**
HKEY_CURRENT_USER **(HKCU) hive, 456, 459, 461–464**
HKEY_LOCAL_MACHINE **(HKLM) hive, 456, 458, 459–461**
HKEY_USERS **hive, 456**
$Home **variable, 61**
Host snapin, 90, 93
$Host **variable, 61**

hyphen (-)
in assignment operator, 220
as subtraction operator, 218, 219
in unary operators, 226–227

I

-ieq **operator, 224**
if **statement, 251–254**
-ige **operator, 224**
-ignoreWhitespace **parameter, 201**
-igt **operator, 224**
-ile **operator, 224**
-ilike **operator, 224**
-ilt **operator, 224**
-imatch **operator, 224**
import-alias **cmdlet, 95, 108, 348**
import-clixml **cmdlet, 95**
import-csv **cmdlet, 95**
-include **parameter**
for clear-variable cmdlet, 231
for get-childitem cmdlet, 191–194
for get-content cmdlet, 197, 200–201
for get-variable cmdlet, 230
for remove-item cmdlet, 167, 173–174
for remove-variable cmdlet, 232
for set-variable cmdlet, 228
for stop-service cmdlet, 178
for test-path cmdlet, 450
-includeChain **parameter, 377**
IndexOf() **method, 265, 277–278**
IndexOfAny() **method, 265, 277–278**
-ine **operator, 224**
-inotlike **operator, 224**
-inotmatch **operator, 224**
$Input **variable, 61**
-InputFormat **PowerShell startup option, 10**
-inputObject **parameter**
for format-table cmdlet, 156
for measure-object cmdlet, 201
for out-file cmdlet, 449
for select-object cmdlet, 145
for where-object cmdlet, 142–143
-inputOption **parameter, for** trace-command **cmdlet, 419**
Insert() **method, 265, 278**
installing
.NET Framework 2.0, 4–7
Windows PowerShell, 7–8, 368
Internet Explorer
closing, 299
as COM application, 294–299
launching, 294–295
manipulating in PowerShell, 294–299
invoke-expression **cmdlet, 95**
invoke-history **cmdlet, 92**
invoke-item **cmdlet, 93**
-is **operator, 79**
IsNormalized() **method, 265**
-isnot **operator, 79**
-isValid **parameter, 451**
-itemType **parameter, 203**

J

join-path **cmdlet, 93, 450, 451–452**

L

-last **parameter, 145, 148–150**
LastIndexOf() **method, 265, 278–279**
LastIndexOfAny() **method, 265, 278–279**
-le **operator, 79, 144, 223**
Length **property, 241, 245, 266, 440**
less-than operator, 79, 144, 223
less-than-or-equal-to operator, 79, 144, 223
-like **operator, 79, 144, 223**
-line **parameter, 201**
-list **parameter, 481**
-listenerOption **parameter, for** trace-command **cmdlet, 419, 420**
-literalName **parameter, 185**
-literalPath **parameter**
for get-childItem cmdlet, 191
for push-location cmdlet, 341
for resolve-path cmdlet, 453
logical operators, 218, 225–226
-logName **parameter, 480**
logs. *See* event logs
looping, 256–261
ls **alias, 52, 78**
-lt **operator, 79, 144, 223**

M

managed resources, 496–497
Management snapin, 90, 93–94
mapping drives, 305–306
-match **operator, 54, 79, 80, 108, 144, 223, 289**

-maximum **parameter, 201**
$MaximumAliasCount **variable, 61**
$MaximumDriveCount **variable, 61**
$MaximumErrorCount **variable, 61, 387**
$MaximumFunctionCount **variable, 61**
$MaximumHistoryCount **variable, 18, 61**
$MaximumVariableCount **variable, 61**
measure-command **cmdlet, 95**
measure-object **cmdlet, 95, 201–203**
MemberInfo **object, 324, 326**
-memberType **parameter, 320, 322, 328, 330**
memory
  Win32_CacheMemory WMI class, 511–512
  Win32_PhysicalMemory WMI class, 510–511
-message **parameter**
  for write-debug cmdlet, 414
  for write-error cmdlet, 400
metacharacter (^), 81, 85, 290, 357, 508
MethodInfo **object, 328, 329**
methods
  for FileInfo object, 436–437
  for finding information about .NET classes, 324–333
  for String class, 264–266
Microsoft Access, as COM application, 303–305
Microsoft Excel, as COM application, 302–303
Microsoft Management Console (MMC), 39
Microsoft Word, as COM application, 301–302
Microsoft.Management.Automation.Core **namespace, 425**
Microsoft.PowerShell.Core **snapin, 90, 91, 92**
Microsoft.PowerShell.Host **snapin, 90, 93**
Microsoft.PowerShell.Management **snapin, 90, 93–94**
Microsoft.PowerShell.Security **snapin, 90, 94**
Microsoft.PowerShell.Utility **snapin, 90, 95–96**
-minimum **parameter, 201**
minus sign (-)
  in assignment operator, 220
  as subtraction operator, 218, 219
  in unary operators, 226–227
move-item **cmdlet, 93**
move-itemProperty **cmdlet, 93**
multiplication operator (*), 218, 219
$MyInvocation **variable, 61**

N

-name **parameter**
  for clear-variable cmdlet, 231
  for get-childItem cmdlet, 191
  for get-member cmdlet, 60, 320, 322
  for get-process cmdlet, 177
  for get-psDrive cmdlet, 185
  for get-variable cmdlet, 230
  for new-item cmdlet, 203
  for new-psDrive cmdlet, 204
  for new-variable cmdlet, 229
  as positional parameter, 133
  for remove-psDrive cmdlet, 187
  for remove-variable cmdlet, 232
  for set-variable cmdlet, 228
  for stop-service cmdlet, 178
  for trace-command cmdlet, 419, 420
named parameters, 124–125
namespaces. *See also* providers
  defined, 338
  as drives, 45–51
  as providers, 346
naming cmdlets, 57–58
-ne **operator, 79, 223**
-neq **operator, 144**
$NestedPromptLevel **variable, 61**
.NET Framework 2.0
  as basis for PowerShell architecture, 34–35
  calling classes, 34
  case sensitivity issue, 18, 326–327
  class library, 36, 37–38
  COM object types, 306–308
  core library, 355–358
  creating .NET objects, 311–319
  downloading Software Developer's Kit, 310
  finding information about .NET classes, 324–333
  future emphasis on, 31
  information resources, 310–311
  installing, 4–7
  matching strings using regular expressions, 54
  PowerShell and, 309–311
  reflection, 324–333
  succinct PowerShell ways to manipulate objects, 56
  type system, 306–308
network shares, mapping drives to local machines, 305–306
new **verb, 57**
new-alias **cmdlet, 95, 108–109, 348**
-newest **parameter, 480**
new-item **cmdlet, 93, 203–204**
new-itemProperty **cmdlet, 93**
new-object **cmdlet**
  -comObject parameter in, 43, 56, 294, 311
  creating new COM objects, 293–294

creating new .NET objects, 311–317
defined, 95
-strict parameter in, 294, 311
-typeName parameter in, 294, 311, 313
new-psDrive **cmdlet, 93, 204–205**
new-service **cmdlet, 93, 357, 360**
new-timespan **cmdlet, 95**
new-variable **cmdlet, 95, 229–230**
-noClobber **parameter, 449**
-NoExit **PowerShell startup option, 10**
-NoLogo **PowerShell startup option, 10**
-noNewLine **parameter, 214**
-NonInteractiver **PowerShell startup option, 10**
nonterminating errors, 381, 382, 392
-NoProfile **PowerShell startup option, 10**
Normalize() **method, 265**
-not **operator, 225**
-notcontains **operator, 79**
Notepad
opening files in, 106
saving .psl scripts in, 22
not-equal-to operator (-ne**), 79, 223**
-notlike **operator, 79, 144, 223**
-notmatch **operator, 79, 144, 223**
$now **variable, 34–37**
$null **variable, 62**
numbers, at beginning of expression, 63–64

O

-object **parameter, 214**
objects, measuring properties, 201–203
-off **parameter, 408**
operators
arithmetic, 217, 218–219
assignment, 72, 217, 220–222
comparison, 217, 222–224
for filtering objects, 79
logical, 218, 225–226
overview, 217–218
precedence, 219–220
redirection, 18, 41, 407, 445–446
special, 218
unary, 218, 226–227
for where-object cmdlet, 144
-option **parameter**
for new-alias cmdlet, 109, 348
for new-variable cmdlet, 229
for set-alias cmdlet, 348
for set-variable cmdlet, 228
for trace-command cmdlet, 419–420
-or **operator, 225**
out-default **cmdlet, 96**
out-file **cmdlet, 96, 449–450, 492**
out-host **cmdlet, 96**
out-null **cmdlet, 96**
out-printer **cmdlet, 96**
-outputBuffer **parameter, 132**
-OutputFormat **PowerShell startup option, 10**
-outputVariable **parameter, 132**
out-string **cmdlet, 96**
OverloadDefinitions **property, 325**

P

PadLeft() **method, 265, 279–280**
PadRight() **method, 265, 279–280**
parameters
absence of, 117, 118
for clear-variable cmdlet, 231
common, 132, 397–399
example, 117
finding for cmdlets, 121–124
for format-table cmdlet, 155–156
for get-authenticodeSignature cmdlet, 378
for get-childItem cmdlet, 191
for get-eventlog cmdlet, 480–481
for get-location cmdlet, 195, 196–197, 338–341
for get-member cmdlet, 320–323
for get-process cmdlet, 118–121
for get-psDrive cmdlet, 185
for get-service cmdlet, 358–360
for get-traceSource cmdlet, 422
for get-variable cmdlet, 230
for get-wmiObject cmdlet, 503
for join-path cmdlet, 451–452
list, 132
for measure-object cmdlet, 201
named, 124–125
for new-item cmdlet, 203
for new-object cmdlet, 293–294, 311–317
for new-psDrive cmdlet, 204
for new-service cmdlet, 360–361
for new-variable cmdlet, 229
for out-file cmdlet, 449–450
overview, 117
for pop-location cmdlet, 344–345
positional, 127–130
for push-location cmdlet, 341–344
for remove-item cmdlet, 166–167
for remove-psDrive cmdlet, 187
for remove-variable cmdlet, 232–233
for resolve-path cmdlet, 453
for restart-service cmdlet, 361

parameters (continued)
for select-object cmdlet, 144
for set-authenticodeSignature cmdlet, 377–378
for set-psDebug cmdlet, 408–413
for set-service cmdlet, 362
for set-traceSource cmdlet, 422
for set-variable cmdlet, 228
for start-service cmdlet, 362–363
for stop-service cmdlet, 363–364
for suspend-service cmdlet, 364
for test-path cmdlet, 450–451
for trace-command cmdlet, 419–420
variables as, 133–135
wildcards in values, 125–127, 200
for write-error cmdlet, 400–401
for write-psDebug cmdlet, 414
parentheses ()
role in parsing, 64
using with methods, 264
parser, PowerShell, 63–69
-passThru **parameter**
for new-variable cmdlet, 229
for pop-location cmdlet, 344, 345
for push-location cmdlet, 341
for set-location cmdlet, 188, 190
for set-variable cmdlet, 228
for stop-service cmdlet, 178
Path **environment variable, 54, 67, 353**
path names
fully qualified, 427–430
in PowerShell, 426–433
relative, 430–431
running commands and, 431–433
-path **parameter**
for copy-item cmdlet, 170
defined, 166
for get-childItem cmdlet, 191
for get-content cmdlet, 196
for join-path cmdlet, 452
for new-item cmdlet, 203
for push-location cmdlet, 341, 344
for remove-item cmdlet, 166, 167
for resolve-path cmdlet, 453
for test-path cmdlet, 450
PathInfo object, 344, 345
paths, cmdlets for, 450–453
-pathType **parameter, for** test-path **cmdlet, 450**
percent sign (%), in assignment operator, 220, 221
period (.)
for running PowerShell scripts, 23, 472
when using PowerShell command mode, 64, 65, 66, 68
$PID **variable, 62**
pipelines (|)
default formatter, 151–154
defined, 78
examples, 13, 14, 19–20, 41–42
grouping objects, 83–85
layout, 19
.NET objects in, 35–38, 57
overview, 19–20, 78–85
past limitations, 56–57
role in sorting objects, 81–83
separators in, 13, 78
sequence of commands, 78–81
symbol, 19, 78
syntax, 19
plus sign (+)
as addition operator, 218, 219, 248
in assignment operators, 220
in unary operators, 226–227
pop-location **cmdlet, 93, 342, 344–345**
positional parameters
overview, 127–131
-property parameter as, 156–157
PowerShell (Windows)
approaches to parsing, 63–69
architecture, 33–34
backward compatibility, 51–56
clearing screen in, 17–18
closing, 10
complete coverage forecast, 55
current working folder, 23
cycling through recently used commands, 18
defined, 3
enabling scripts, 9–10
error handling in, 381–382
exiting, 10
exploring Windows systems, 69–76
extended wildcards, 59–60
extensibility, 51–56
finding available commands, 11–14, 75–76
installing, 7–8, 368
loading console files, 90
loading snapins, 90–91
long term roadmap, 55
minimizing default risk, 368–374
need for, 25–31
.NET Framework basis, 34–35, 309–311
as object-based, 35–58
path names in, 426–433
repeating last-used command, 18
response to errors, 58, 66–67

starting, 8–11, 89–90
support for code debugging, 59
synthetic types, 306–308
system state information, 338–350
unsigned, 23
upgrade path to C#, 58
using to explore Windows registry, 458–461
working with file system, 425–453
PowerShell Analyzer, 41
`PowerShell` **command, 89–90**
precedence, operator, 219–220
preference variables, 115–116
`-process` **parameter, 261**
profile files, 97–100
`$Profile` **variable, 62**
`profile.ps1` **file, 98**
`$ProgressPreference` **variable, 62**
`prompt` **function, 113**
`-prompt` **parameter, 212, 213, 216**
prompts. *See* command prompt
properties
expanding, 146–147
for `FileInfo` object, 438–439, 440, 441
measuring, 201–203
selecting, 145–146
`-property` **parameter**
for `format-table` cmdlet, 155, 156–157
for `measure-object` cmdlet, 201
for `select-object` cmdlet, 145–146
`PropertyInfo` **object, 330, 331**
`PropertyItem` **class, 152–153**
providers. *See also* drives; `get-psProvider` **cmdlet**
defined, 45, 183
finding, 183–184, 425
in fully qualified path names, 427
namespaces as, 346
overview, 45–51
relationship to drives, 45, 186
removing items, 166–175
WMI, 499
`.pscl` **extension, 90**
`-PSConsoleFile` **PowerShell startup option, 10**
`-psDrive` **parameter, 195, 338, 339**
`$PSHome` **variable, 62**
`-psHost` **parameter, for** `trace-command` **cmdlet, 419**
`.psl` **extension, 22**
`-psProvider` **parameter**
for `get-location` cmdlet, 195, 338, 339
for `get-psDrive` cmdlet, 185
for `new-psDrive` cmdlet, 204
for `remove-psDrive` cmdlet, 187
`push-location` **cmdlet, 94, 341–344**
`$PWD` **variable, 62**

Q

question mark (?)
in `$?` variable, 61
as wildcard, 59, 72, 125, 171
quotation mark, double (")
at beginning of expression, 64, 65
in commands, 71, 431–432
parameters and, 119–120
quotation mark, single ('), 119, 416, 431–432

R

`-readCount` **parameter, 196**
`read-host` **cmdlet, 96, 134–135, 211, 212–214**
`-recommendedAction` **parameter, 400**
`-recurse` **parameter**
for `copy-item` cmdlet, 170
for `get-childItem` cmdlet, 191
for `new-item` cmdlet, 167
for `remove-item` cmdlet, 166, 173–174
redirection operator (>>), 445–446
redirection operator (>), 18, 41, 407, 445–446
reflection, .NET, 324–333
`RegEdit` **utility. *See* Registry Editor**
registry
adding new keys, 457–458
allowed types, 457
backing up, 456, 457
list of hives, 456
making changes to, 461–464
navigating by using PowerShell, 458–461
overview, 455–458
removing items, 166–175
Registry Editor
accessing registry keys, 308
modifying `ExecutionPolicy` key value, 209–211
running, 456
`Registry` **provider, 47–48, 184, 185, 346**
regular expressions, 289–291
relative path names, 430–431
remote machines, 513–514
`RemoteSigned` **execution policy, 208, 209, 211, 370, 372, 374**
`Remove()` **method, 266, 280–281**
`remove-item` **cmdlet, 94, 166–175**
`remove-itemProperty` **cmdlet, 94**
`remove-psDrive` **cmdlet, 94, 187**
`remove-psSnapin` **cmdlet, 92**

remove-variable **cmdlet, 96, 232–233**
rename-item **cmdlet, 94**
rename-itemProperty **cmdlet, 94**
Replace() **method, 266, 281–282**
$ReportErrorShowExceptionClass **variable, 62**
$ReportErrorShowInnerException **variable, 62**
$ReportErrorShowSource **variable, 62**
$ReportErrorShowStackTrace **variable, 62**
repository, CIM, 499
-resolve **parameter, for** join-path **cmdlet, 452**
resolve-path **cmdlet, 94, 450, 453**
restart-service **cmdlet, 94, 357, 361, 364**
Restricted **execution policy, 207, 208, 209, 370, 372, 373**
resume-service **cmdlet, 94, 357**
-root **parameter, 204**

S

-scope **parameter**
  for clear-variable cmdlet, 231
  for get-psDrive cmdlet, 185
  for get-variable cmdlet, 230
  for new-psDrive cmdlet, 204
  for new-variable cmdlet, 229
  for remove-psDrive cmdlet, 187
  for remove-variable cmdlet, 233
  for set-variable cmdlet, 228
Script Library, WMI, 497–498
scripts, PowerShell
  avoiding aliases in, 87
  for COM objects, 43–44
  enabling, 9–10, 68, 207–217
  execution policies, 207–212, 370
  minimizing default risk, 368–374
  overview, 21–23, 41–43
  path issues, 22, 23
  path names and, 431–432
  running, 22–23, 368, 431–432
  saving .ps1 scripts in Notepad, 22
  signing, 376–379
  testing cmdlet combinations, 21–23
scroll bars, 18
security
  execution policies and, 370–374
  minimizing default risk, 368–374
  running scripts, 368–374
Security log, 478
Security snapin, 90, 94
select-object **cmdlet**
  defined, 96, 144
  -expandProperty parameter in, 146–147
  finding positional parameters, 127–128
  -first parameter in, 148–150
  -last parameter in, 148–150
  overview, 144–145
  properties, 144–145
  -property parameter in, 145–146
  for selecting most recent event log entries, 487–488
  specifying properties to pass along, 104
  -unique parameter in, 147–148
  using to explore files, 439–442
select-string **cmdlet, 96**
semicolon (;), 20, 198
services, list of cmdlets, 357
Services MMC snap-in. *See* get-service **cmdlet**
set **verb, 57**
set-acl **cmdlet, 94**
set-alias **cmdlet, 55, 96, 109–111, 348**
set-authenticodeSignature **cmdlet, 94, 376, 377**
set-content **cmdlet, 22, 94**
set-date **cmdlet, 96**
set-executionPolicy **cmdlet, 9, 94, 371**
set-item **cmdlet, 94**
set-itemProperty **cmdlet, 94**
set-location **cmdlet, 46, 47–48, 49, 94, 188–190**
set-psDebug **cmdlet, 92, 408–413, 415, 417**
set-service **cmdlet, 94, 357, 362**
set-traceSource **cmdlet, 96, 422**
set-variable **cmdlet, 96, 228–229**
shell. *See* Windows PowerShell
shell aliases, system state information, 338, 346–349
shell functions, system state information, 338, 349–350
shell variables, system state information, 338, 350
$ShellId **variable, 62**
-showErrors **parameter, 156**
single quotation mark ('), 119, 416, 431–432
signed scripts, 376–379
slash (/)
  in assignment operator, 220, 221
  in division operator, 218, 219
snapins
  loading, 90–91
  Microsoft.PowerShell.Core snapin, 90, 91, 92
  Microsoft.PowerShell.Host snapin, 90, 93
  Microsoft.PowerShell.Management snapin, 90, 93–94
  Microsoft.PowerShell.Security snapin, 90, 94
  Microsoft.PowerShell.Utility snapin, 90, 95–96
  viewing cmdlets available, 91–92
sorting pipeline objects, 81–83

`sort-object` **cmdlet, 81–83, 96, 104, 484**
spelling mistakes, 349
`Split()` **method, 266, 282–285**
`split-path` **cmdlet, 94, 450**
square brackets ([])
in PowerShell syntax for indicating .NET elements, 34, 268, 324
in syntax for designating array elements, 236
in wildcard searches, 59, 290, 429
`-stack` **parameter**
for `get-location` cmdlet, 195, 338
for `push-location` cmdlet, 341, 343
`-stackName` **parameter**
for `get-location` cmdlet, 195, 338
for `push-location` cmdlet, 341, 343, 344
starting Windows PowerShell, 8–10, 89–90
`start-service` **cmdlet, 94, 357, 362–363**
`start-sleep` **cmdlet, 96**
`StartsWith()` **method, 266, 285**
`start-transcript` **cmdlet, 93**
startup options, PowerShell, 10–11
`-static` **parameter, 320, 322**
`-step` **parameter, 408, 411–413**
`stop-process` **cmdlet, 94, 175–178**
`stop-service` **cmdlet, 94, 178–181, 357, 363–364**
`stop-transcript` **cmdlet, 41, 93**
`StoreCountAndDate.ps1` **script, 22**
`-strict` **parameter, 294, 311**
strings. *See also* `System.String` **class**
casting to other classes, 287–291
comparing, 268–270
copying, 267
`Substring()` **method, 266**
subtraction operator (-), 218, 219
`-sum` **parameter, 201**
`Suspend` **option, 412, 413**
`suspend-service` **cmdlet, 94, 357, 364**
`switch` **statement, 254–256**
syntax errors, 403–408
synthetic types, 306–308
system errors, 381–401
System log, 478
system state, 338–350
`System.AppDomain` **class, 353, 354**
`System.Boolean` **object, 315**
`System.Collections.HashTable` **object, 249**
`System.____ComObject` **type, 306–307**
`System.DateTime` **class, 34, 35. *See also*** `DateTime` **object**
`System.Diagnostics.Process` **object, 34**
`System.Management.Automation.Cmdlet` **class, 381**
`System.Management.Automation.Core` **namespace, 185**
`System.Management.Automation.PSCmdlet` **class, 381**
`System.Reflection` **namespace, 324**
`System.Resources` **namespace, 356–357**
`System.Security.SecureString` **object, 213–214**
`System.String` **class**
`Length` property, 266
list of methods, 265–266
overview, 263–264
working with methods, 267–287
`System.Type` **class, 333**

T

Tab key, using to complete commands, 76–77, 112–113, 443–444
`-targetObject` **parameter, 400**
`tee-object` **cmdlet, 96**
terminating errors, 381, 382
`test-path` **cmdlet, 94, 450–451**
`ThrowTerminatingError()` **method, 381**
time and date examples, 34–38
`-timeStampServer` **parameter, 377**
`ToCharArray()` **method, 266, 285–286**
`ToLower()` **method, 266, 286–287**
`ToLowerInvariant()` **method, 266, 286–287**
`ToString()` **method, 266**
`-totalCount` **parameter, 197, 198**
`ToUpper()` **method, 266, 287**
`ToUpperInvariant()` **method, 266, 287**
`-trace` **parameter, 408–410**
`trace-command` **cmdlet**
defined, 96, 419
examples, 420–421
multiple positional parameters, 129–131
parameters for, 419–420
tracing, 418–423
`trap` **statement, 392–397, 410–411**
`Trim()` **method, 266, 287**
`TrimEnd()` **method, 266, 287**
`TrimStart()` **method, 266, 287**
`$True` **variable, 62**
typed arrays, 239–240
`-typeName` **parameter, 294, 311, 313**

U

unary operators (- and –), 226–227
underscore (`$_`**) variable, 61, 71**
`-unique` **parameter, 145, 147–148**

Unrestricted **execution policy, 208, 209, 370, 372, 373, 374**
update-formatData **cmdlet, 96, 162**
update-typeData **cmdlet, 96, 162**
URI **object, 287–288**
Utility snapin, 90, 95–96

V

-value **parameter**
for new-item cmdlet, 203
for new-variable cmdlet, 229
for set-variable cmdlet, 228
-valueOnly **parameter, 230**
Variable **provider, 184, 185, 346**
variables
assigning values to, 228–229, 352
automatic, 60–62
clearing value, 231–232
creating, 229–230
as drives, 49, 346, 350
environment, 351–353
error-related, 345–346, 383–392
manipulating, 227–233
overview, 49–51
as parameters, 133–135
preference, 115–116
removing, 232–233
retrieving, 230–231
-verbose **parameter**
as common parameter, 132, 165, 166
for stop-service cmdlet, 181–182
$VerbosePreference **variable, 62**
verbosity, 85–87
verbs, in cmdlets, 13
-view **parameter, 156**
visible **property, 295–296**
Visual Studio 2005, 310
vertical bar (|)
default formatter, 151–154
defined, 78
examples, 13, 14, 19–20, 41–42
grouping objects, 83–85
layout, 19
.NET objects in, 35–38, 57
overview, 19–20, 78–85
past limitations, 56–57
role in sorting objects, 81–83
separators in, 13, 78
sequence of commands, 78–81
symbol, 19, 78
syntax, 19

W

-wait **parameter, 197, 199, 200**
$WarningPreference **variable, 62**
WBEM (Web-based enterprise management), 30
-whatIf **parameter**
for clear-variable cmdlet, 231
as common parameter, 166
defined, 132, 165
for new-item cmdlet, 203
for new-psDrive cmdlet, 204
for new-variable cmdlet, 229
overview, 39
for remove-item cmdlet, 170–171
for remove-psDrive cmdlet, 187
for remove-variable cmdlet, 233
set-authenticodeSignature cmdlet and, 377
for set-executionPolicy cmdlet, 371
for set-variable cmdlet, 228
stop-process cmdlet and, 175–178
stop-service cmdlet and, 178–180
$WhatIfPreference **variable, 62**
where-object **cmdlet**
defined, 92, 137
for filtering event log entries, 484–487, 489, 490
filtering overview, 138–140
-filterScript parameter in, 142–143
finding aliases for, 138, 347–348
-inputObject parameter in, 142–143
list of operators, 144
-match operator and, 290–291, 347
multiple filter tests, 138–140
operators for use in filtering, 79–81, 144
overview, 41, 137–138
parameters for, 142–143
simple filtering, 138, 139–140
together with GetProperties() method, 330
using to filter processes, 71–72, 73
using to filter services, 74–75
while **statement, 258–259**
-width **parameter, 449**
wildcards
asterisk as, 12, 48, 49, 59, 72, 73, 125, 171–173, 200, 202
extended, 59–60
for filtering processes, 72–73
in parameter values, 125–127, 200
question mark as, 59, 72, 125, 171
for removing items, 171–173
Win32_CacheMemory **WMI class, 511–512**
Win32_Date **WMI class, 497**

Win32_PhysicalMemory **WMI class, 510–511**
Win32_Process **WMI class, 497, 508–509**
Win32_Processor **WMI class, 497, 509–510**
Win32_Service **WMI class, 497, 512–513**
windir **environment variable, 465–466**
Windows Explorer, 368
Windows Management Instrumentation (WMI)
accessing, 56
CIM Studio tool, 499–500
defined, 30–31
managed resources layer, 496–497
overview, 496–502
relationship to PowerShell, 38–39, 495
role of aliases, 51–52
tools, 499–502
WMI consumers layer, 499
WMI infrastructure layer, 497–499
WMI Object Browser, 500–502
Windows PowerShell
approaches to parsing, 63–69
architecture, 33–34
backward compatibility, 51–56
clearing screen in, 17–18
closing, 10
complete coverage forecast, 55
current working folder, 23
cycling through recently used commands, 18
defined, 3
enabling scripts, 9–10
error handling in, 381–382
exiting, 10
exploring Windows systems, 69–76
extended wildcards, 59–60
extensibility, 51–56
finding available commands, 11–14, 75–76
installing, 7–8, 368
loading console files, 90
loading snapins, 90–91
long term roadmap, 55
minimizing default risk, 368–374
need for, 25–31
.NET Framework basis, 34–35, 309–311
as object-based, 35–58
path names in, 426–433
repeating last-used command, 18
response to errors, 58, 66–67
starting, 8–11, 89–90
support for code debugging, 59
synthetic types, 306–308
system state information, 338–350
unsigned, 23
upgrade path to C#, 58
using to explore Windows registry, 458–461
working with file system, 425–453
Windows Script Host (WSH), 30, 299–301
Windows systems
exploring processes with get-process cmdlet, 69–71
exploring services with get-service cmdlet, 73–75
exploring with get-wmiobject cmdlet, 509–514
filtering processes by using wildcards, 72–73
filtering processes with where-object cmdlet, 69–71
finding running processes by using get-process cmdlet, 69–71
WMI consumers, 499
WMI Object Browser, 500–502
WMI providers, 499
WMI Script Library, 497–498
WMI (Windows Management Instrumentation)
accessing, 56
CIM Studio tool, 499–500
defined, 30–31
managed resources layer, 496–497
overview, 496–502
relationship to PowerShell, 38–39, 495
role of aliases, 51–52
tools, 499–502
WMI consumers layer, 499
WMI infrastructure layer, 497–499
WMI Object Browser, 500–502
Word, as COM application, 301–302
-word **parameter, 201**
-wrap **parameter, 155**
write-debug **cmdlet, 96, 413–418**
write-error **cmdlet, 96, 396, 400–401**
WriteError() **method, 381**
write-host **cmdlet**
defined, 96
examples, 211, 394–395, 418
overview, 214–217
using to display expression results, 66
versus write-debug cmdlet, 413
write-output **cmdlet, 96**
write-progress **cmdlet, 96**
write-verbose **cmdlet, 96**
write-warning **cmdlet, 96**
WSH (Windows Script Host), 30, 299–301

X

XML files, console files as, 90
XML **object, 289**