

Pro Windows PowerShell



Hristo Deshev

Pro Windows PowerShell

Copyright © 2008 by Hristo Deshev

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-940-2

ISBN-10 (pbk): 1-59059-940-3

ISBN-13 (electronic): 978-1-4302-0546-3

ISBN-10 (electronic): 1-4302-0546-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Tony Campbell

Technical Reviewer: Jon Rolfe

Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell,
Jonathan Gennick, Kevin Goff, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann,
Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Sofia Marchant

Copy Editor: Heather Lang

Associate Production Director: Kari Brooks-Copony

Production Editor: Laura Cheu

Compositor: Susan Glinert Stevens

Proofreader: Linda Seifert

Indexer: Julie Grady

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com>. You will need to answer questions pertaining to this book in order to successfully download the code.

To my wife, Yana, for all her love and support

Contents at a Glance

About the Author	xvii
About the Technical Reviewer	xix
Acknowledgments	xxi
Introduction	xxiii

CHAPTER 1	Objects and Object Types	1
CHAPTER 2	Controlling Execution Flow	33
CHAPTER 3	The Object Pipeline	55
CHAPTER 4	Working with Script Blocks	69
CHAPTER 5	Working with Functions	87
CHAPTER 6	Command Aliases	107
CHAPTER 7	Providers	127
CHAPTER 8	Script Files	139
CHAPTER 9	Error Handling and Debugging	155
CHAPTER 10	Signing Scripts	197
CHAPTER 11	The Shell Environment and Its Configuration	219
CHAPTER 12	Extending the Type System	237
CHAPTER 13	Getting Help	253
CHAPTER 14	Taming Processes and Services	275
CHAPTER 15	Input and Output	285
CHAPTER 16	Monitoring Your System	297
CHAPTER 17	PowerShell and the World Wide Web	315
CHAPTER 18	Sending E-mail	349
CHAPTER 19	Talking to COM Objects	367
CHAPTER 20	Managing Windows with WMI	395
CHAPTER 21	PowerShell Community Extensions	417

■ CHAPTER 22	PSEventing: .NET Events in PowerShell	449
■ CHAPTER 23	Enhancing Tab Completion with PowerTab	461
■ INDEX		473

Contents

About the Author	xvii
About the Technical Reviewer	xix
Acknowledgments	xxi
Introduction	xxiii
CHAPTER 1 Objects and Object Types	1
The Extended Type System	1
Accessing Object Properties	2
Using Object Methods	2
Object Adapters	3
Type Extensions	5
Built-in Types	8
Strings and String Operations	8
Numeric Types	14
Arrays and Collections	16
Dictionaries and Hash Tables	21
Type Literals	25
Type Conversion	26
Accessing Static Members	30
Summary	31
CHAPTER 2 Controlling Execution Flow	33
Conditional Expressions	33
Comparing Values	34
Implicit Type Conversions	36
Logic and Bitwise Operations	37
Boolean Conversions	41
The Power of String Conversions	42
Collections and Conditional Expressions	43

Branching Instructions	44
Simple Branching: if/else	44
Multiple Branches and the switch Statement	45
Loops: Repeatedly Executing Blocks of Code	48
The Simplest Loop: while	48
Loops and Counters: for Loop	50
Executing Actions on All Objects in a Collection: foreach Loop	50
Controlling Loop Execution: break and continue Statements	51
Summary	54
 CHAPTER 3 The Object Pipeline	 55
Text-Based Pipelines	55
Object-Based Pipelines	56
Processing Collections Using ForEach-Object	59
Filtering Collections Using Where-Object	60
Adding or Deleting Properties Using Select-Object	60
Sorting Collections	62
Pipeline Tees	63
Grouping Objects	64
Gathering Collection Statistics	65
Detecting Changes and Differences Among Objects	65
The Object Pipeline and Functional Programming	66
Summary	67
 CHAPTER 4 Working with Script Blocks	 69
Defining Script Blocks	69
Passing Parameters and Returning Values	71
Processing Pipeline Input	75
Variable Scope	77
Invoking Strings as Expressions	83
Script Blocks as Delegates	85
Summary	86
 CHAPTER 5 Working with Functions	 87
Defining Functions	87
Function Internals	88
Function Parameters	89
Passing Parameters by Reference	91

Returning Values	93
Scoping Rules	94
Filters	97
Functions and Script Blocks	101
Implementing New Control Structures	101
Script Blocks as Strategies	104
Summary	106

■ CHAPTER 6 **Command Aliases** 107

Working with Aliases	107
Creating Aliases	108
Modifying Aliases	110
Exporting and Importing Aliases	111
Aliasing Tips, Techniques, and Pitfalls	115
Name Clashes	115
Complex Aliases	117
Removing Broken Aliases	118
Built-in Aliases	119
cmd.exe Look-alikes	120
UNIX Look-alikes	121
Convenience Aliases	123
Summary	125

■ CHAPTER 7 **Providers** 127

Enumerating Providers	127
Drives	129
Drive Scope	131
Navigating to Drives	132
Provider Capabilities	134
Basic Provider Capabilities	134
Drive Providers	134
Item Providers	134
Item Container Providers	135
Navigation Providers	136
Item Content Providers	136
Item Property Providers	136
Dynamic Item Property Providers	137
Item Security Descriptor Providers	137
Summary	137

CHAPTER 8	Script Files	139
	Creating Scripts	139
	Invoking Scripts	140
	Passing Parameters	142
	Returning Values	146
	Executing PowerShell Scripts from Other Environments	148
	Developing and Maintaining Script Libraries	149
	Dot-sourcing as a Means to Include Script Libraries	150
	The Library Path Problem	150
	Summary	154
CHAPTER 9	Error Handling and Debugging	155
	Handling Errors	155
	Common Error-Handling Parameters	157
	Trapping Errors	160
	Capturing Nonterminating Errors	170
	Raising Errors	172
	Debugging Your Code	177
	Print Debugging	178
	Stepping Through Scripts and Breaking Execution	188
	Tracing Script Execution Details	190
	Summary	196
CHAPTER 10	Signing Scripts	197
	How Code Signing Works	197
	Guaranteeing Script Integrity	198
	Certifying the Origin of a Script	198
	Managing Certificates	199
	Creating a Self-Signed Certificate	202
	Creating a Certification Authority Certificate	202
	Issuing a Code-Signing Certificate	206
	Signing Scripts	211
	Running Scripts on Other Machines	215
	Summary	217

CHAPTER 11	The Shell Environment and Its Configuration	219
	Shell Hosts	219
	User Profile Scripts	223
	Settings for All Users and All Shells	223
	Settings for All Users Using a Specific Shell	224
	Settings for a Specific User and All Shells	224
	Settings for a Specific User and a Specific Shell	224
	Working with Saved Console Settings	226
	Changing the Prompt Settings	228
	Tab Expansion: How Command Completion Works	231
	Summary	235
CHAPTER 12	Extending the Type System	237
	Modifying Objects and Types	237
	Adding Members to Single Objects	238
	Adding Members to All Instances of a Class	244
	Extending Object Formatting	247
	Creating Our Own Views	249
	Summary	252
CHAPTER 13	Getting Help	253
	The Help System	253
	The Basics	254
	Parameter Details	256
	Advanced Techniques	258
	Getting Command Information	264
	Getting Information About Objects	266
	Using the Internet to Get Help	268
	Summary	273
CHAPTER 14	Taming Processes and Services	275
	Working with Processes	275
	Listing Processes: Getting the Big Picture	276
	Starting and Stopping Processes	277
	Processes and Their Windows	278
	Process Modules: On What Libraries Does This Baby Depend?	279

Getting Information About a Program's Publisher	280
Setting Process Priority	280
Administering Services	281
Starting and Stopping Services	282
Configuring Services	283
Analyzing Service Dependencies	283
Summary	284
 ■ CHAPTER 15 Input and Output	285
Reading Content	285
Writing Content	288
Content Encoding	289
Getting Byte Content	290
Different Unicode Encodings	291
Extracting Data from Text	294
Finding Matches for a Regular Expression	294
Finding String Occurrences Inside Files	295
Summary	296
 ■ CHAPTER 16 Monitoring Your System	297
Working with the Windows Event Log	297
Reading from the System Event Logs	298
Writing to the Event Log	302
Performance Counters	306
Consuming Counter Data	307
Monitoring Programs	309
Summary	313
 ■ CHAPTER 17 PowerShell and the World Wide Web	315
Laying the Foundation	315
Fetching Files from the Web	316
Setting Connection Options and Debugging	
Connection Problems	318
Testing and Validating Web Sites	328
Test-Url: Verifying a Page's Existence	328
Checking If a Page Contains Broken Links	330

Working with News Feeds	332
Getting Feeds	333
Calling Web Services	338
Calling Web Services Using HTTP GET Requests	338
Calling Web Services Using HTTP POST Requests	340
Calling Web Services Using the SOAP Protocol	342
Handling FTP Transactions	345
Downloading Files from an FTP Server	345
Uploading Files to an FTP Server	346
Summary	347
 CHAPTER 18 Sending E-mail	349
Using System.Net.Mail	349
Building a Reusable Script	350
Configuring Recipients	352
Working with Message Headers	355
Multipart Messages	357
Message Views	357
Attaching Files to Messages	359
Embedding Media in Messages	361
Putting It All Together	363
Summary	365
 CHAPTER 19 Talking to COM Objects	367
How COM Works	368
How PowerShell Supports COM	368
Scripting Programs Through Their COM Interfaces	369
Automating Microsoft Word	369
Scripting Microsoft Excel	378
Driving Internet Explorer	384
Windows Script Host Code Interoperability	389
Evaluating Code	390
Exposing Objects from MSScriptControl	392
Summary	393

CHAPTER 20	Managing Windows with WMI	395
	A Brief History of WMI	395
	WBEM and WMI Components	396
	The Common Information Model	396
	Network Transport	397
	Object Discovery	397
	PowerShell's WMI Support	398
	Get-WmiObject: The WMI Query Tool	399
	Language Support for WMI Objects	401
	Exploring CIM Classes and Objects	404
	Getting a List of Classes and Namespaces	404
	WMI System Properties	406
	Querying Hardware Devices	407
	Getting Information About Software	408
	Operating System Updates	408
	Installed Programs	409
	The Network Configuration	411
	Generating WMI Code	411
	Scriptomatic	412
	Summary	414
CHAPTER 21	PowerShell Community Extensions	417
	Installation and Configuration	417
	Tab Expansion	418
	Editing Configuration Files	418
	Getting Help	419
	File System Utilities	421
	Navigation Helpers	421
	NTFS Helpers	422
	Compressing and Archiving Files	424
	Network Utilities	425
	Executing Processes and Commands	427
	XML Tools	430
	Working with Image Files	435
	Clipboard Helpers	437

New Providers	440
Reading from the .NET Global Assembly Cache	440
Exploiting the Internet Explorer Feed Store	441
Manipulating Windows Active Directory	443
Utility Applications	446
Summary	448
 CHAPTER 22 PSEventing: .NET Events in PowerShell	449
Introducing PSEventing	449
Event-Handling Mechanics	450
FileSystemWatcher: A Real-World Example	451
Monitoring Entries Written to the System Event Logs	452
Handling WMI Events	454
Detecting If Our Script Has Been Terminated by the User	456
Using Script Blocks As Event Handlers	457
Summary	459
 CHAPTER 23 Enhancing Tab Completion with PowerTab	461
Installation	462
How PowerTab Works	463
Data Grid Pop-up Windows	466
Configuring PowerTab	467
The IntelliSense Completion Handler	468
The Tab Expansion Database	470
Summary	471
 INDEX	473

About the Author

HRISTO DESHEV has been a software developer, a team leader, and a product manager for Telerik, the leading vendor of user interface components (<http://www.telerik.com>). For the past several years, his focus has been on creating rich and, at the same time, robust and reliable web and desktop components running on the Microsoft .NET platform. A major goal for him and his team is to predictably deliver working software—applying agile software engineering practices to create rock-solid solutions targeting all modern web browsers and supporting multiple Visual Studio .NET versions.

Hristo is a reformed UNIX fanatic who now runs Windows and tries to apply his scripting skills and experiences to the Microsoft platform. He is also a tool freak: anything that can help automate a task—be it a scriptable utility or a full-blown scripting language—is a welcome addition to his toolbox. PowerShell has been a passion of his since its early unofficial releases due to its ease of use and almost incredible ability to automate all Windows-related tasks without needing other tools.

You can reach Hristo through his blog at <http://weblogs.asp.net/hristodeshev/>.

About the Technical Reviewer

■ **JON ROLFE** has been an IT professional for more than 12 years and is currently a senior solutions architect for one of the world's largest IT services companies. He specializes in designing secure Microsoft-based enterprise architectures for clients in the United Kingdom's public sector and large blue-chip companies. During the course of his career, he has gained extensive experience at all stages of the IT life cycle, including enterprise architecture design, software development, rapid system deployment, and large enterprise systems support. Although he specializes in Microsoft technologies, Jon also has an interest in other technologies including Linux and wrote the book *Using Ubuntu Linux* (Bernard Babani, 2007).

When not working, Jon has an active interest in digital photography, follows motor sports, and enjoys mountain biking and skiing. He can be contacted through his web site at <http://JonRolfe.com>.

Acknowledgments

I still find it hard to believe that I wrote an entire book! It has been an enjoyable project that was also quite formidable and downright hard. I would never have done it all on my own, without the huge amounts of help I got from many people.

First, I would like to thank the entire Apress team that guided me through the process: Tony Campbell for providing important input and valuable feedback on the content, Sofia Marchant for being so flexible with my sometimes late chapters, Heather Lang for her extraordinary ability to turn my often unreadable writings into well-thought-out text. Jon Rolfe deserves a special mention here; he has been an awesome technical reviewer. He not only spotted a host of problems with the sample scripts but was keen to provide solutions and improvement ideas; thus his efforts became one of the most important factors in getting this book done.

I would also like to thank the Microsoft PowerShell team for creating such a wonderful product. Being a long-time UNIX user, I have been disappointed by the primitive Windows command line shell. With PowerShell, this is no longer the case.

The PowerShell community deserves special recognition too. The wealth of information and thoughtful discussions on blogs and online forums has been an incredible inspiration when writing this book.

Last, but not least, I am personally thankful to the people behind the open source tools covered in the book: Keith Hill and the numerous contributors to the PowerShell Community Extensions project; Marc van Orsouw, the creator of the PowerTab tab completion extension; and Oisín Grehan, the man behind the PowerShell Eventing Library and many other PowerShell-related open source projects. You guys rock!

Introduction

I remember the first time I got to play with Windows PowerShell—I had some spare time on my hands, and after reading a blog post, I quickly set off to download the unofficial pre-release bits that were available at the time. I originally had very low expectations about the tool—all I wanted was a shell that is up to par with its UNIX cousins. Boy, was I wrong! The moment I learned about object pipelines, I knew this tool was going to change the way I think about automating tasks on Windows. Fast forward a couple of months, and here I am, so much in love with PowerShell that I even wrote a whopping 400-page book about it.

Why Another Book on Windows PowerShell?

As much as I like PowerShell, its plain-console looks make it very easy for people to dismiss it without really seeing what lies beneath the surface. I feel it is still largely undiscovered—I know many professionals who have yet to try it out. When administrators see PowerShell for the first time, many simply shrug, assuming it's just another version of the primitive command prompt we inherited from DOS.

The most successful way for me to recruit a new person into the ranks of happy PowerShell users is by example. When presented with an automation problem, my first suggestion is, “Let’s try to do it in PowerShell.” A couple of lines later, the usual reaction is, “Wow, can it really do that?” A short and sweet command can win many hearts for PowerShell. Hopefully, the examples in this book will win your heart too—it is my goal to share with you the joy of working with a modern automation environment that will change your professional life forever.

What’s in This Book?

When working on the book, I separated it into sections, and this distinction may be helpful to you too.

The first section includes Chapters 1 through 14. It introduces the shell’s object-oriented features and its means for providing basic abstraction like script blocks, functions, scripts, aliases, and providers. It is best to read those chapters in sequence to get a solid understanding of the advanced techniques that allow you to work with the shell’s type system, get to know its security infrastructure, learn about the documentation facilities, and start using important debugging techniques.

The second section spans Chapters 15 to 21. The text uses advanced scripting techniques to work with .NET, COM, and WMI objects. This section’s goal is twofold. On one hand, you’ll learn how to work with external code and employ those techniques to automate almost any task on a Windows system. On the other hand, you are solving real-world problems that are very likely to land on your plate some time soon. The chapters describe how to start and stop processes

and services, work with text-based I/O, use performance counters to monitor your system, download and upload files to the web, send e-mail, automate programs via their COM interfaces, and get information and manage operating system objects through WMI. The chapters can be read in random order, and you can easily use them for reference.

The last section of the book is about tools and add-ons that you can use to extend the shell and become even more productive. Many programs can get much easier to use after extending them with a tool that provides an important missing feature. The book shows how you can use freely available open source tools to boost your productivity. An important part of my secret in winning over new people for PowerShell lies in the fact that I always do my demonstrations with PowerShell Community Extensions and PowerTab installed. Those two tools are described in Chapters 21 and 23 respectively.

Do I Need Prior Programming Experience?

The book is written by a programmer—does that mean you need to be a programmer to read it? Not at all! I have done my best to make the text easy to digest for just about anyone with a little prior scripting experience. The PowerShell scripting language bears a lot of similarities to other scripting languages, and users will be able to draw parallels between it and VBScript, JScript, batch scripts, UNIX shell scripts, and maybe even Perl or Python. People new to scripting should be able to just use any of the working code samples as they are, without needing to learn the details or any of the inner workings of the code.



Objects and Object Types

Starting a journey in an unknown programming language and an unfamiliar environment requires that we start with the fundamentals. What are the most basic, the absolutely essential parts of any environment? Variables of course! Variables are those tiny virtual boxes that hold our data and let us mold it into whatever we like. PowerShell is unusual in that all variables hold references to objects. All objects conform to a known, published contract; an object contains a set of operations, usually called methods, which allow us to execute actions. Additionally, PowerShell objects expose properties that allow us to get and set object attributes.

PowerShell is based on the Microsoft .NET framework, but a fully featured shell would go further than supporting .NET objects only. PowerShell does exactly that through its flexible type system. It adapts and extends objects of different origins, so that we, the users, can not only work with them but do so in a uniform way. Learning about the type system and getting to know the most commonly used types is the most important step on the road to PowerShell mastery.

The Extended Type System

Rule number one for PowerShell is that everything is an object. Objects can have different types and origins and can contain various data too. Nevertheless, they must all look the same and expose services in a similar fashion, so that shell scripters do not need to learn different syntaxes for different objects. The first and most important characteristic of an object is its type. A type holds information about the operations that an object supports and is most often a .NET class. To get an object type, we use the `GetType()` method that all .NET objects have:

```
PS C:\> (42).GetType()
```

IsPublic	IsSerial	Name	BaseType
-----	-----	----	-----
True	True	Int32	System.ValueType

```
PS C:\> "Hello, world".GetType()
```

IsPublic	IsSerial	Name	BaseType
-----	-----	----	-----
True	True	String	System.Object

```
PS C:\> (1,2,3).GetType()
```

IsPublic	IsSerial	Name	BaseType
-----	-----	-----	-----
True	True	Object[]	System.Array

Accessing Object Properties

An object type contains object members. Members are usually properties and methods. Technically, they can be fields or events, but we will rarely see those in scripts. Properties are generally used to expose object data, and guess what? Data, stored in a property, is usually another object. We can access property values using the following dot notation:

```
PS C:\> $user = @{}
PS C:\> $user.Name = "John"
PS C:\> $user.Name
John
```

The preceding example creates a new dictionary object and stores it in the `$user` variable. The code then assigns the “John” string as the `Name` property value. The last line gets the property value—as you can see, it holds the same string we just put in there.

Properties can be read only or read-write, depending on whether their values can be modified. Of course, the shell will complain if we try to assign a new value to a read-only property:

```
PS C:\> "Hello".Length = 7
"Length" is a ReadOnly property.
At line:1 char:9
+ "Hello".L <<<< ength = 7
```

Strings are immutable objects, so to change a string’s length, we would have to create a new string object.

Many objects contain special properties, called indexer properties. Those properties expose collections of objects and allow accessing them via a name or an index. Most often we use that with an array:

```
PS C:\> $fruit = "apples", "oranges"
PS C:\> $fruit[0]
apples
PS C:\> $fruit[1]
oranges
```

Using Object Methods

An object method is a piece of code that represents an operation that the object supports. It usually takes parameters and returns other objects. As an example, here is how we can check if a string contains another string:

```
PS C:\> "Hello".Contains("Hell")
True
```

Methods do not necessarily take parameters. Here is the syntax that converts a string object to uppercase by calling its `ToUpper` method:

```
PS C:\> "uppercase, please".ToUpper()
UPPERCASE, PLEASE
```

Note that we still need to add the parentheses at the end even if we do not need to pass parameters for the method. Forgetting the parentheses will make PowerShell think we are trying to get hold of the method object itself:

```
PS C:\> "uppercase, please".ToUpper
```

```
MemberType      : Method
OverloadDefinitions : {System.String ToUpper(), System.String ToUpper(CultureInfo culture)}
TypeNameOfValue  : System.Management.Automation.PSMethod
Value           : System.String ToUpper(), System.String ToUpper(CultureInfo culture)
Name            : ToUpper
IsInstance      : True
```

That may be of use to us if we wish to get more information about the method itself, but most often we just want to call it.

Object Adapters

.NET objects are not the only kids in the object-oriented game. A self-respecting shell would allow us to use various application's automation objects, Active Directory data, and Windows Management Instrumentation classes and objects at the least—which all have different origins and are implemented using different technologies. We need a mechanism that will make those look and work just like .NET objects do. PowerShell calls that mechanism object adaptation. Internally, the shell knows how to work with the various object types, and that knowledge is spread among a set of adapter objects. When a foreign object comes along, the shell wraps it in a native .NET object that serves as a view of the original object. The view object is of the `PSObject` type object and uses the adapter to get properties and methods. Most of the time, a `PSObject` behaves just like the original object, and the user can hardly tell he or she is using a special object. If we need the original object, we can always get it using the special `PSBase` property. Now, here is a list of the adapters the shell uses when it works with objects.

ManagementObjectAdapter

This adapter knows about Windows Management Instrumentation (WMI) objects and makes working with them easier by, for example, exposing their `Properties` collection as object properties. Here is a taste of the clumsy syntax that we would have had to use to get a process's caption if we did not have that adapter:

```
PS C:\> (Get-WmiObject win32_process)[0].PSBase.Properties["Caption"].Value
System Idle Process
```

Note the use of the `PSBase` property to get to the raw, unadapted object. Using the adapted version, we can just say:

```
PS C:\> (Get-WmiObject win32_process)[0].Caption
System Idle Process
```

DirectoryEntryAdapter

This adapter works with the Windows active directory in a similar way to the `ManagementObjectAdapter`. Here is how it hides the ugliness of working with a nested `Properties` collection:

```
$user = [ADSI]"WinNT://./Hristo,user"
$user.PSBase.Properties["Name"]
```

behind a normal property:

```
PS C:\> $user = [ADSI]"WinNT://./Hristo,user"
PS C:\> $user.Name
Hristo
```

DataRowAdapter

This class knows about the ADO.NET `DataRow` object and exposes row properties accessible via the column names as object properties. Let's take as an example a data table saved as XML in the following format:

```
<Users>
  <User>
    <Name>John</Name>
    <Age>25</Age>
  </User>
  <User>
    <Name>Mike</Name>
    <Age>20</Age>
  </User>
</Users>
```

Here is how we access row data:

```
PS C:\> $ds = New-Object Data.DataSet
PS C:\> $ds.ReadXml("C:\PowerShell\data.xml")
InferSchema
PS C:\> $ds.Tables[0].Rows[0].Name
John
PS C:\> $ds.Tables[0].Rows[0].PSBase["Name"]
John
```

The last line uses the raw row object as an illustration of what row data access would have looked like without the adapter.

DataRowViewAdapter

This guy works in the same way as `DataRowAdapter`—the only difference is that it adapts ADO.NET `DataRowView` objects.

XmlNodeAdapter

This very important adapter class transfers inner objects and XML attributes as object properties to make things easier for the user. The .NET XML classes make it all too complicated by making the user think if a value is stored as an XML attribute or as an element value. The adapter makes all values look like properties to PowerShell code. Here is how we can navigate our XML-based user data by using the property syntax:

```
PS C:\> $xmlDoc = New-Object Xml.XmlDocument
PS C:\> $xmlDoc.Load("C:\PowerShell\data.xml")
PS C:\> $xmlDoc.Users.User[0]
```

Name	Age
----	---
John	25

Convenient, isn't it? Exposing inner elements as properties allows us to quickly query an XML tree without using heavy-duty tools like XPath at all. XML finally made easy!

ComAdapter

Working with COM objects involves dealing with the compiled type information and accessing the COM type libraries. The PowerShell COM adapter knows how to do that and spares us the details. Look how the Internet Explorer COM object works just like any other object you have seen so far:

```
PS C:\> $ie = New-Object -COM InternetExplorer.Application
PS C:\> $ie.Visible = $true
```

Type Extensions

Object adapters are the primary mechanism of changing how an object looks, but they are internal to PowerShell: they are implemented as .NET objects and are an integral part of the PowerShell code base. Since there is no way for a user to create his or her own adapter or extend an existing one, we have another mechanism of modifying object types: type extensions. The type system allows users to provide additional code and data and add members, both properties and methods, to objects and object types at runtime. PowerShell supports extending objects by adding many different types of members, and we will go through doing that in greater detail in Chapter 12. For now, this is how you can get some of the extended properties of the `System.Diagnostics.Process` type:

```
PS C:\> (Get-Process)[0] | Get-Member -type AliasProperty
```

```
TypeName: System.Diagnostics.Process
```

Name	MemberType	Definition
----	-----	-----
Handles	AliasProperty	Handles = Handlecount
Name	AliasProperty	Name = ProcessName
NPM	AliasProperty	NPM = NonpagedSystemMemorySize
PM	AliasProperty	PM = PagedMemorySize
VM	AliasProperty	VM = VirtualMemorySize
WS	AliasProperty	WS = WorkingSet

The code gets the first of all processes on the system and displays its alias properties. You can see that PowerShell extends the `Process` type, so that we can use shorter names for some of the properties it exposes: `Name` instead of `ProcessName`, `WS` instead of `WorkingSet`, and so on.

Alias properties just reference other properties by name and return their values. Property aliasing is usually done to shorten some of the most commonly used properties to save typing and to make objects look consistent. For example, one of my pet peeves with .NET has always been that collections have a `Count` property that returns the number of items inside. Arrays, on the other hand, do not have a `Count` property—they have a `Length` property instead. I have to remember which property to use all the time. PowerShell solves that problem by adding an extended `Count` property to arrays, relieving me of this burden:

```
PS C:\> Get-Member -input (1,2,3) -type AliasProperty
```

```
TypeName: System.Object[]
```

Name	MemberType	Definition
----	-----	-----
Count	AliasProperty	Count = Length

Remember how the `PSBase` special property gave us access to the raw unadapted object? We can get access to the extended object view only by accessing the `PSExtended` special property in much the same manner. This is how to get all the extended members for the `Process` type:

```
PS C:\> (Get-Process)[0].PSExtended | Get-Member
```

```
TypeName: System.Management.Automation.PSMemberSet
```

Name	MemberType	Definition
----	-----	-----
Handles	AliasProperty	Handles = Handlecount
Name	AliasProperty	Name = ProcessName
NPM	AliasProperty	NPM = NonpagedSystemMemorySize
PM	AliasProperty	PM = PagedMemorySize
VM	AliasProperty	VM = VirtualMemorySize
WS	AliasProperty	WS = WorkingSet
__NounName	NoteProperty	System.String __NounName=Process
PSConfiguration	PropertySet	PSConfiguration {Name, Id, PriorityClass,...
PSResources	PropertySet	PSResources {Name, Id, Handlecount, Worki...

```

Company      ScriptProperty System.Object Company {get=$this.Mainmodu...
CPU          ScriptProperty System.Object CPU {get=$this.TotalProcess...
Description  ScriptProperty System.Object Description {get=$this.Main...
FileVersion  ScriptProperty System.Object FileVersion {get=$this.Main...
Path         ScriptProperty System.Object Path {get=$this.Mainmodule....
Product      ScriptProperty System.Object Product {get=$this.Mainmodu...
ProductVersion ScriptProperty System.Object ProductVersion {get=$this.M...

```

Again, we will go through all types of extended members in Chapter 12. For now, just think about Note properties, as a way to return a constant value from a property, and for Script properties, as a way to attach our own script code that will get executed whenever a property is gotten or set. Going back to our array extension and the Count property, this is how to get to it:

```
PS C:\> (1,2,3).PSExtended | Get-Member
```

```
TypeName: System.Management.Automation.PSMemberSet
```

```

Name MemberType Definition
---- -
Count AliasProperty Count = Length

```

You might have noticed that the PSExtended property returns a special type of object: PSMemberSet. Member sets are logical groups of members that we can use to create different views of objects. To get the member sets that are defined for an object, we can again use the Get-Member command and indicate explicitly that we are interested in PropertySet and MemberSet property types:

```
PS C:\> (Get-Process)[0] | Get-Member -type PropertySet,MemberSet
```

```
TypeName: System.Diagnostics.Process
```

```

Name MemberType Definition
---- -
PSConfiguration PropertySet PSConfiguration {Name, Id, PriorityClass, Fi...
PSResources      PropertySet PSResources {Name, Id, Handlecount, WorkingS...

```

In the preceding example, you can see two views defined for a Process object: one, PSConfiguration, that focuses on how a process has been configured and another, PSResources, that reports on the system resources that are being used. We can use those property sets when displaying information about an object using one of the Format-* commands:

```
PS C:\> Get-Process WinWord | Format-List PSResources
```

```

Name      : WINWORD
Id        : 3368
HandleCount : 356
WorkingSet : 44331008

```

```
PagedMemorySize      : 20971520
PrivateMemorySize    : 20971520
VirtualMemorySize    : 331403264
TotalProcessorTime   : 00:03:12.5937500
```

```
PS C:\> Get-Process WinWord | Format-List PSConfiguration
```

```
Name           : WINWORD
Id              : 3368
PriorityClass    : Normal
FileVersion     : 11.0.8134
```

As a summary, each object has a set of views that it must support. They are accessible through special properties:

- **PSObject**: The default view is what we get when we work with the object directly.
- **PSBase**: The raw object view that gives us access to a .NET or COM object that has not been adapted by the type system.
- **PSAdapted**: The adapted object view has members added or filtered out after the object has been processed by its adapter.
- **PSExtended**: The extended view of the object contains only members added as type extensions.

Built-in Types

PowerShell not only allows us access to many types of objects and lets us modify them in a uniform way—it also includes several built-in types that are worth knowing, as they are the ones we will be using in our everyday work with the shell. To be honest, PowerShell does not build those types from scratch; it relies heavily on the .NET framework, and all the built-in types are really .NET types. The shell designers have extended those types and added special syntax shortcuts to the language to make it easier to use them.

Strings and String Operations

Character strings are arguably the most commonly used data type in any programming language. Most likely that is the case because strings contain free-form data and are used to transport data across system boundaries. PowerShell's string support is built on top of the .NET `System.String` object type. That means that a string in PowerShell will have all operations that .NET strings have. On top of that, strings have a number of useful extensions that increase the expressiveness and effectiveness of string manipulation.

String Literals

Creating string literals is easy; we just need to surround a sequence of characters in single or double quotation marks:

```
PS C:\> "This is a string"
This is a string
PS C:\> 'This is a string too'
This is a string too
```

Why support both types of quotes? That is a convenience feature. Mixing quotes allows us to embed quotation marks inside strings without having to escape them:

```
PS C:\> 'John said: "OK."'
John said: "OK."
PS C:\> "Let's go"
Let's go
```

Sometimes, strings get complex, and mixing types of quotation marks becomes a burden. In that case, we can just escape a quotation mark and move along. To do so, simply type the symbol twice:

```
PS C:\> "John said: ""OK.""
John said: "OK."
PS C:\> 'Let''s go'
Let's go
```

Another method of escaping quotes is to use the general escape symbol, the backtick (`):

```
PS C:\> "John said: `\"OK.`"
John said: "OK."
```

One thing to keep in mind is that backtick escaping works in double-quoted strings only. The backtick can be used as an escape character for various symbols that are hard to type:

- ``0`: The null symbol is used as a string end delimiter. It does not have a printable representation.
- ``a`: The alert symbol is the character having the 7 ASCII character code. When a string that contains that character gets printed on the screen, it will produce a beep on the computer speaker. The symbol itself will have no visible representation.
- ``b`: Backspace will delete the previous symbol:

```
PS C:\> "aa`b bb"
a bb
```

- ``f`: The form feed or page break symbol will make a printer start printing on a new page. Again, it has no printable representation.

- ``n`: The line feed symbol causes a printer or terminal to start printing on a new line:

```
PS C:\> "aa`nbb"
aa
bb
```

- ``r`: The carriage return symbol causes a printer or terminal to start printing on the beginning of the line. The character will not generate a new line, and subsequent characters will be printed at the beginning of the current line possibly overwriting previous output. In Windows systems, this symbol is used in conjunction with ``n` as a line separator in text files. The ``r`n` sequence indicates a new line.
- ``t`: The horizontal tab symbol moves several symbols forward, according to the current console settings:

```
PS C:\> "aa`tbb"
aa      bb
```

- ``v`: Though the vertical tab symbol is rarely, it may prove useful when doing printing or formatting console-based output.

A useful feature of the backtick symbol is that you can use it to escape the new line symbol when typing at the console. Ending your line with a backtick signals to the shell that the command continues on the next line:

```
PS C:\> Get-Item C:\Windows\System32\WindowsPowerShell\v1.0\types.ps1xml `
>> | Get-Content | Measure-Object
>>
```

```
Count      : 3131
Average    :
Sum        :
Maximum    :
Minimum    :
Property   :
```

Often, we want to input a large block of text without having to take care of escaping quotes. This is where here strings come handy! Here strings are special markers that indicate that everything between them is a part of the string. To create such a string, we surround it with `@"` and `"@"` or `@'` and `'@`:

```
PS C:\> @"
>> line one
>>   line two
>>
>> line three
>> "@
>>
line one
   line two
```

```

line three
PS C:\> '@'
>> "Let's
>>     go downtown!", John said.
>> '@
>>
>> "Let's
>>     go downtown!", John said.

```

Note that all white space, like new lines and tabs, is preserved in here strings. Keep in mind that the string start and end markers *must* be placed alone on separate lines.

String Interpolation

One of the greatest productivity boosters when working with strings is string interpolation, a widely used feature, also known as variable substitution, that is present in many script and shell languages. In essence, these terms refer to the ability to nest variable names in string literals and have the shell expand those variables and embed their values instead. Typically, we use that feature to format messages for display or to craft specially formatted strings before passing them to other programs. Here is how we can send a message to the user:

```

PS C:\> $processCount = (Get-Process).Count
PS C:\> "$processCount processes running in the system."
61 processes running in the system.

```

Sometimes, we do not want to treat a dollar-sign expression as a variable name. To prevent that, we have to escape the dollar sign using a backtick:

```

PS C:\> "`$processCount is the property that we need."
$processCount is the property that we need.

```

Another option is to use a single-quoted string. Interpolation only works on double-quoted strings:

```

PS C:\> '$processCount processes running in the system.'
$processCount processes running in the system.

```

String interpolation works for here strings too:

```

PS C:\> @"
>> Processes
>> -----
>> $processCount
>> "@
>>
>> Processes
>> -----
>> 61

```

Analogous to ordinary single-quoted strings, there is no interpolation when using single-quoted here strings.

We can use some really complex variable expressions inside strings, but we have to be careful and obey the formatting rules. Variable parsing stops at the end of the first word, and to use an expression, we have to put it inside a `$()` block:

```
PS C:\> $processes = (Get-Process)
PS C:\> "$($processes.Count) processes running in the system."
60 processes running in the system.
```

Note that variables inside the `$()` block still need to be prefixed with dollar signs. Now, let's go a step further and get rid of the `$processes` variable entirely:

```
PS C:\> "$((Get-Process).Count) processes running in the system."
60 processes running in the system.
```

We can embed any legal PowerShell expression:

```
PS C:\> "Total due: $(12 * 5000)"
Total due: 60000
```

We can even cause side effects when generating strings and modify variable values from embedded expressions:

```
PS C:\> $times = 0
PS C:\> "Operation performed $($times++; $times) times"
Operation performed 1 times
PS C:\> $times
1
```

Caution While useful at times, relying on side effects triggered by string interpolations is usually considered bad programming practice and is best avoided. Most people expect that variable expressions do not modify the system state, and doing the opposite will lead to many hard-to-track bugs.

Productivity Boosting String Operators

In PowerShell, strings have been extended with special syntax so that commonly used operations are easier to perform. PowerShell uses the familiar dash notation that looks like providing a command parameter. The most important operations follow.

Formatting

Instead of calling the static `String.Format` method like this:

```
PS C:\> [string]::Format("{0} running processes.", $processCount)
61 running processes.
```

PowerShell can format a string once it sees the `-f` parameter:

```
PS C:\> "{0} running processes." -f $processCount
61 running processes.
```


Wildcard Matching

This is similar to matching using a regular expression without all the intrinsic power regular expressions have. The advantage to this method is that it is really intuitive and simple to use. For example, we can test if a string starts with the word “Test” as follows:

```
PS C:\> "Test string" -like "Test*"
True
PS C:\> "Sample string" -like "Test*"
False
```

The asterisk symbol means “zero or more symbols of any type.” We can use a question mark as a match for exactly one symbol of any type:

```
PS C:\> "notepad.exe" -like "notepad.???"
True
PS C:\> "notepad.exe" -like "notepad.?"
False
PS C:\> "notepad.exe" -like "?otepad.exe"
True
```

We can also use a character range. Here is how to check if a string starts with any letter and ends with “s” or “n”:

```
PS C:\> "notepads" -like "[a-z]*[sn]"
True
PS C:\> "notepadz" -like "[a-z]*[sn]"
False
PS C:\> "_notepads" -like "[a-z]*[sn]"
False
```

Regular Expression Matching

Regular expression matching is similar to wildcard matching with the only difference being that we use the `-match` operator and provide a regular expression pattern. Regular expressions are a vast subject on which entire books have been written, so I will not be covering them here. As an example, this is how you match something that looks like a domain name:

```
PS C:\> "www.yahoo.com" -match "(www\.)?w+\.(com|org|net)"
True
PS C:\> "yahoo.com" -match "(www\.)?w+\.(com|org|net)"
True
PS C:\> "yahoo.org" -match "(www\.)?w+\.(com|org|net)"
True
```

The expression matches strings that optionally start with “www. ”, followed by one or more alphanumeric characters, followed by a dot and ending in “com”, “org”, or “net”. Note that the dot (.) symbol has a special meaning in regular expressions. It signifies any character, so we escape it with a backslash.

Tip For more information you can visit the excellent online resource that is the <http://www.regular-expressions.info> site and go through its tutorials and references on the subject. The site covers regular expression usage in different programming languages and environments. While the syntax and support are more or less the same on most platforms, there are some subtle differences, so make sure you go through the article about .NET regular expressions at <http://www.regular-expressions.info/dotnet.html>. PowerShell uses the .NET regular expressions under the hood.

Replacing Strings or Substrings

PowerShell provides the `-replace` operator to replace string occurrences. This operator behaves much differently than the `.NET String.Replace()` method! `String.Replace()` takes a string as a parameter and replaces all occurrences in the target string:

```
PS C:\> "one.test, two!test".Replace(".test", "-->DONE")
one-->DONE, two!test
```

`-replace` takes a regular expression and internally calls `Regex.Replace()`. Be careful when providing the replace pattern, and escape regular expression special characters if needed. The preceding example might not work as expected if we use `-replace`:

```
PS C:\> "one.test, two!test" -replace ".test", "-->DONE"
one-->DONE, two-->DONE
```

The dot instructs the regular expression engine to look for any character, so our exclamation mark gets replaced too. To correct that, we have to escape the dot:

```
PS C:\> "one.test, two!test" -replace "\.test", "-->DONE"
one-->DONE, two!test
```

Having interpolation and regular expressions under your belt will significantly boost your scripting productivity, so practicing those two until they feel natural might be an excellent idea.

Numeric Types

PowerShell supports all native .NET numeric types. Probably the only ones that we will eventually need to construct from literals are those in the following sections.

System.Int32

`System.Int32`, also known as `int`, is a 32-bit integer value that is the default for most operations. Creating one is a matter of typing a sequence of digits:

```
PS C:\> (3).GetType().FullName
System.Int32
```


Yes, my winword process is using more than 20MB of virtual memory. In fact, it is eating up about 310 of them. You can use a similar technique for files and other objects in your computer:

```
PS C:\> (Get-Item C:\pagefile.sys -Force).Length / 1GB
1.5
```

Pagefile.sys is a system file, so I had to use the -Force parameter to get to it.

Numbers as Strings

Finally in this section we will look at how numbers can be formatted and converted to strings. We do that by using the -f string operator. Some of the particularly useful formatting options follow:

- *Currency*: Use a {0:c} format string. The currency symbol will be obtained from the current culture.

```
PS C:\> "{0:c}" -f 2.5
$2.50
```

- *Percentage*: The format string is {0:p}:

```
PS C:\> "{0:p}" -f 0.2
20.00 %
```

- *Fixed number of digits*: Use the pound sign (#) as a digit placeholder. For example, this line will format a number and round it to the third digit after the decimal point:

```
PS C:\> "{0:#.###}" -f 3.4567
3.457
```

- *Scientific notation*: Some people find scientific notation odd; others, useful. Passing a {0:e} format string will get us a properly formatted normalized mantissa and an exponent:

```
PS C:\> "{0:e}" -f 333.4567
3.334567e+002
```

There are a number of other formats available in .NET that we will not cover here. To get all of them, peruse the System.Globalization.NumberFormatInfo class documentation on MSDN.

Arrays and Collections

Arrays and other ordered collections play a very important part in all programming tasks. I will use the words “array” and “ordered collection” interchangeably in this section, as PowerShell adapts all list collections so that they work in the same way, and we cannot tell one type of collection from another (it is possible to do so, but there is little value in that).

Creating an array is as simple as listing all its members and separating them with commas. Here is a small array containing numbers:

```
PS C:\> 1, 2, 3, 4
1
2
```

```
3
4
```

```
PS C:\> (1, 2, 3, 4).GetType().FullName
System.Object[]
```

Note that the array is of the `Object` type, which means we can hold all types of objects inside:

```
PS C:\> 1, 2.5, "apples", (Get-Process winword)
1
2.5
apples
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
352	15	19804	46920	311	112.83	3252	WINWORD

```
PS C:\> (1, 2.5, "apples", (Get-Process winword)).Count
4
```

The preceding code creates an array of objects that contains an integer, a double precision floating point number, a string, and a `Process` object—that's four types of objects in total: an `int`, a `double`, a `string`, and a `System.Diagnostics.Process` instance.

Array creation has a very free-form syntax. The most formal syntax is to wrap the members in a `@()` block:

```
PS C:\> @("one", "two")
one
two
```

This is the only way to create an empty array:

```
PS C:\> (@()).Count
0
```

```
PS C:\> (@()).GetType().FullName
System.Object[]
```

As a convenience, arrays can be created using the numeric range notation. It requires that we provide the start and end numbers and fills in the values between them for us:

```
PS C:\> 5..1
5
4
3
2
1
```

```
PS C:\> 1..5
```

```
1
2
3
4
5
```

```
PS C:\> (0..255).Count
256
```

That syntax is extremely powerful for enumerating ports, addresses, and all sorts of sequential data.

The array creation syntax recognizes nested arrays and flattens them into one. This allows us to add arrays and even ranges in place:

```
PS C:\> 1, (5, 6), 2
1
5
6
2
```

```
PS C:\> 1, (10..7), 2
1
10
9
8
7
2
```

We can access the array items by using the bracket notation:

```
PS C:\> $a = 2,3,4
PS C:\> $a[0]
2
PS C:\> $a[1]
3
PS C:\> $a[2] = 5
PS C:\> $a
2
3
5
```

Note that we modify an item by directly assigning a new value to it. PowerShell arrays, just like the .NET ones, are zero-based, which means that the first item is always at index zero, and the last one at the item count minus one.

Interestingly enough, we can pass more than one index inside the brackets. The return is an array that contains the items at those indexes. This is called an array slice. Here is how to get the first and the last item from an array:

```
PS C:\> $a = 2,3,4
PS C:\> $a[0,2]
2
4
```

The preceding example is a bit problematic, because it requires us to know that the array has three items, so that we request the one at index 2. Many programming languages, PowerShell included, allow us to address items starting from the end by using negative indexes. Here is how to change the previous example, so that our code does not have to use a hard-coded number for the array length in order to get the last value:

```
PS C:\> $a = 2,3,4
PS C:\> $a[0,-1]
2
4
```

We can create slices by passing ranges as our array indexes too! Here is how to get five items, starting from the third:

```
PS C:\> $a = (1, 2, 3, 4, 5, 6, 7, 8)
PS C:\> $a[2..6]
3
4
5
6
7
```

Note that the ranges are inclusive, so we have to pass 2..6, instead of 2..7, to get the five items.

Assigning arrays to items will flatten the arrays. This makes for a very convenient way to insert several items at a specific location:

```
PS C:\> $a = 1, 2, 3
PS C:\> $a[1] = 10, 11, 12
PS C:\> $a
1
10
11
12
3
```

Note that the original value of the second item got destroyed. To preserve it, we need to put forth some extra effort. Prepending the inserted array with the original value does the trick:

```
PS C:\> $a = 1, 2, 3
PS C:\> $a[1] = $a[1], 10, 11, 12
PS C:\> $a
1
2
10
11
```

```
12
```

```
3
```

Similar to strings, arrays have the plus operator defined for concatenation. “Adding” an object or another array to an array will result in a new array that contains all values:

```
PS C:\> $a = 1,2
```

```
PS C:\> $a = $a + 3
```

```
PS C:\> $a
```

```
1
```

```
2
```

```
3
```

The preceding operation adds an item to a variable and assigns the result to the same variable, effectively modifying the variable in place; it is so common that most languages define a shortcut operator (+=) that does exactly that. PowerShell supports that too:

```
PS C:\> $a += 4
```

```
PS C:\> $a
```

```
1
```

```
2
```

```
3
```

```
4
```

```
PS C:\> $a += 5,6
```

```
PS C:\> $a
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

Searching for items inside collections is a routine task in many languages. How do we search an array for a specific value? Here is an attempt:

```
foreach ($item in (2,3,4)){ if ($item -eq 3){ echo "Found" }}
```

That feels too clumsy, and we have to repeat the same loop everywhere we need to search for an array item! Checking if an array contains an item is so common that the PowerShell designers have provided two specific operators for that: -contains and -notcontains:

```
PS C:\> (2, 3, 4) -contains 3
```

```
True
```

```
PS C:\> (2, 3, 4) -notcontains 3
```

```
False
```

```
PS C:\> (2, 3, 4) -notcontains 5
```

```
True
```


Of course, those operators work for different variable types as well:

```
PS C:\> (2, "some value", 4) -contains "some value"
True
```

Beware of the built-in type conversions though! You may get something quite different than what you expect:

```
PS C:\> (2, 3, 4) -contains "3"
True
PS C:\> (2, 3, 4) -contains "3.0"
True
```

See how the “3.0” string got converted to the integer 3 and signaled True? If you need to look for strings and avoid type coercion, you may have to use a loop or the Where-Object command:

```
PS C:\> (2, 3, 4) | Where-Object { $_ -is [string] -and $_ -eq "3.0"}
PS C:\> (2, "3.0", 4) | Where-Object { $_ -is [string] -and $_ -eq "3.0"}
3.0
```

I will describe loops in the next chapter and talk about the Where-Object command in greater detail in Chapter 3.

Dictionaries and Hash Tables

Dictionaries, also called associative arrays, are objects that map a set of keys to a set of values. They contain a number of key-value pairs and allow for easy lookup of values given their keys. Dictionaries are a convenient way to store related data that usually comprises a logical record about a real-world entity. The syntax for creating a dictionary in PowerShell looks like the formal array creation syntax; the only difference is that we use braces. Here is a sample personal record dictionary that has two items with keys “Name” and “Address” respectively:

```
PS C:\> $d = @{"Name"="John"; "Address"="12 Easy St."}
PS C:\> $d
```

Name	Value
----	-----
Name	John
Address	12 Easy St.

PowerShell uses `System.Collection.Hashtable` objects internally whenever it creates a dictionary object. `Hashtable` implements the .NET `IDictionary` interface, and PowerShell’s extended type system allows us to work in the same way with all `IDictionary` objects.

```
PS C:\> $d.GetType().FullName
System.Collections.Hashtable
```

Dictionaries do not impose any restrictions on the type of objects that we use as keys or values. Here is how to add different objects to a `Hashtable`:

```
PS C:\> $d = @{"Name"="John"; "Age"=30; `
>>      "File"=(Get-Item C:\PowerShell\Test.txt)}
>>
PS C:\> $d
```

Name	Value
----	-----
Name	John
Age	30
File	C:\PowerShell\Test.txt

\$d contains a string, an int, and a FileInfo object. Our dictionary is starting to look like a person's record taken from a real database already!

You are not restricted to using only strings as dictionary keys. We can certainly use numbers, but in general, any object will do. Assuming we have Windows Calculator and Microsoft Word running on the system, here is how to create a set of Process objects and map them to string values:

```
PS C:\> $set = @{(Get-Process winword)="MS Word"; (Get-Process calc)="calc"}

PS C:\> $set
```

Name	Value
----	-----
System.Diagnostics.Process ...	calc
System.Diagnostics.Process ...	MS Word

Empty dictionaries are created with the @{ } expression:

```
PS C:\> $empty = @{ }
PS C:\> $empty.GetType().FullName
System.Collections.Hashtable
```

There are several ways to access values in a dictionary. The most important one is the array-like notation; we provide the key name in brackets:

```
PS C:\> $d = @{"Name"="John"; "Age"=30; "Address"= "12 Easy St."}
PS C:\> $d["Name"]
John
```

Dictionaries can behave like objects, and we can access their values as properties using the dot notation:

```
PS C:\> $d = @{"Name"="John"; "Age"=30; "Address"= "12 Easy St."}
PS C:\> $d.Name
John
```

The object notation is more limiting than the array one in a sense that it is hard to provide a key that contains spaces, new lines, or other fancy symbols. The key also has to be convertible from a string. To provide free-form key values, you have to use the array notation.

The object notation supports some fancy ways of providing key values too. This is how to use a string as the property name:

```
PS C:\> $d."Name"
John
```

We can use a variable as the key:

```
PS C:\> $property = "Name"
PS C:\> $d.$property
John
```

and even an expression:

```
PS C:\> $d.($property)
John
PS C:\> $d.("Na" + "me")
John
```

Similar to array slices, dictionaries support getting many values at once by providing an array of keys in the brackets. The return object is an array that contains the values. This is how to get only a person's name and address from our previous array:

```
PS C:\> $d = @{"Name"="John"; "Age"=30; "Address"= "12 Easy St."}
PS C:\> $d["Name", "Address"]
John
12 Easy St.
```

Adding items to a dictionary is a just matter of assigning a new key-value pair with either the object or array notation:

```
PS C:\> $d.Department = "Accounting"
PS C:\> $d["SSN"] = 123456789
PS C:\> $d
```

Name	Value
-----	-----
Department	Accounting
Name	John
Age	30
SSN	123456789
Address	12 Easy St.

Removing items is a bit trickier. We have to use the `Remove()` method and provide the key:

```
PS C:\> $d.Remove("Age")
PS C:\> $d.Remove("SSN")
PS C:\> $d
```

Name	Value
-----	-----
Department	Accounting
Name	John
Address	12 Easy St.

Other important dictionary methods that are worth knowing are the ones that verify if an item is present. `Contains()` and `ContainsKey()` are synonyms that check if an item with the given key exists in the dictionary:

```
PS C:\> $d.Contains("John")
False
PS C:\> $d.Contains("Name")
True
PS C:\> $d.ContainsKey("Name")
True
```

`ContainsValue()` is our friend when we want to look for a value:

```
PS C:\> $d.ContainsValue("John")
True
PS C:\> $d.ContainsValue("Name")
False
```

In addition, dictionaries expose all their keys and values as arrays through the `Keys` and `Values` properties:

```
PS C:\> $d.Keys
Department
Name
Address
PS C:\> $d.Values
Accounting
John
12 Easy St.
```

We can even substitute our calls to `ContainsKey()` and `ContainsValue()` with array searches:

```
PS C:\> $d.Keys -contains "Name"
True
PS C:\> $d.Values -contains "John"
True
```

Remember that `-contains` will silently convert types under the hood! `ContainsKey()` and `ContainsValue()` do not.

Of course, just like any .NET collection, dictionaries have a `Count` property that tells us the number of key-value pairs stored inside:

```
PS C:\> $d = @{"Name"="John"; "Age"=30; "Address"= "12 Easy St."}
PS C:\> $d.Count
3
```

Type Literals

The PowerShell language allows us to access types by using a special syntax called type literals. A type literal is just a type name enclosed in brackets. At their very simplest, type literals return the underlying .NET System.Type object instance:

```
PS C:\> [System.Int32]
```

IsPublic	IsSerial	Name	BaseType
True	True	Int32	System.ValueType

```
PS C:\> [System.String]
```

IsPublic	IsSerial	Name	BaseType
True	True	String	System.Object

```
PS C:\> [System.Diagnostics.Process]
```

IsPublic	IsSerial	Name	BaseType
True	False	Process	System.Compon...

We do not really need to repeat that `System` namespace prefix all the time. Most of the types that we work with are below the `System` namespace, and PowerShell already knows that. It allows us to omit the prefix altogether:

```
PS C:\> [Diagnostics.Process]
```

IsPublic	IsSerial	Name	BaseType
True	False	Process	System.Compon...

We also have a number of type aliases defined for our convenience. The most popular follow:

- `int`, `short`, `long` for referencing the `System.Int32`, `System.Int16` and `System.Int64` types respectively.
- `byte` and `sbyte` for the unsigned `System.Byte` and the signed `System.SByte` types.
- `bool` for `System.Boolean`.
- `void` for no type at all.
- `string` for the `System.String` type.
- `float` and `single` for `System.Single` and `double` for `System.Double`.

- `decimal` aliases `System.Decimal`.
- `regex` points to `System.Text.RegularExpressions.Regex`.
- `adsis` refers to `System.DirectoryServices.DirectoryEntry`.
- `wmi`, `wmiclass`, and `wmsearcher` refer to `System.Management.ManagementObject`, `System.Management.ManagementClass`, and `System.Management.ManagementObjectSearcher`, respectively.

You do not need to remember what each alias stands for; you can always check:

```
PS C:\> ([regex]).FullName
System.Text.RegularExpressions.Regex
```

```
PS C:\> ([adsis]).FullName
System.DirectoryServices.DirectoryEntry
```

```
PS C:\> ([wmsearcher]).FullName
System.Management.ManagementObjectSearcher
```

```
PS C:\> ([wmi]).FullName
System.Management.ManagementObject
```

Type Conversion

Most often, type literals are used to instruct the shell to convert an object from one type to another. The syntax involved requires that we place a type literal before the original object, and the operation will return a new object if nothing goes wrong. Here is how to convert a string to an integer:

```
PS C:\> $str = "10"
PS C:\> $str.GetType().FullName
System.String
PS C:\> $num = [int] $str
PS C:\> $num
10
PS C:\> $num.GetType().FullName
System.Int32
```

Of course, things can and will go wrong. There are a lot of conversions that are just impossible. In that case, an exception will be thrown:

```
PS C:\> [int] "not an integer string"
Cannot convert value "not an integer string" to type "System.Int32". Error:
    "Input string was not in a correct format."
At line:1 char:6
+ [int] <<<< "not an integer string"
```

How PowerShell Converts Between Types

PowerShell really does its best when trying to convert between types. It relies heavily on the .NET framework type conversion mechanisms and exposes a mechanism that programmers can use to provide conversion facilities. Here are the strategies the shell tries when tasked with a conversion operation:

1. It checks if the new type is directly assignable from the old. For that to be true in .NET, the types have to be the same or one of them must inherit directly or indirectly from the other.
2. Well-known ways to convert from one object to another have been built into the language itself; PowerShell next attempts to convert using these. See the “Built-in Type Conversion Rules” section for details.
3. Looks for a `TypeConverter` or a `PSTypeConverter` associated class. `TypeConverter` classes are a part of the .NET native mechanism of converting types, and `PSTypeConverter` is a PowerShell-specific converter object. Both `TypeConverter` and `PSTypeConverter` classes can be associated with types at compile time, by marking our types with the correct .NET attributes. Many existing .NET classes already are associated with type converters, and PowerShell will seamlessly convert them to and from other types. Additionally, we can associate a type with a type converter in the shell’s extended type system configuration files.
4. PowerShell next tries to convert the original object to a string and to find and call a `Parse()` method on the new object type that will create a new object from a string representation.
5. It then looks for a constructor on the new type that will accept an object of the original type. If it finds one, it uses it to construct the new object.
6. After that, PowerShell looks for both explicit and implicit type cast operators on both object types. Appropriate cast operators are compiled as static methods with the names `op_Explicit` and `op_Implicit` for explicit and implicit casts, respectively. If the shell finds an appropriate method, it calls the method and uses the returned object.
7. Finally, PowerShell checks if the original type implements the `IConvertible` .NET interface. If it does, then the type can be converted to the basic types like `int`, `double`, `decimal`, and so on.

Built-in Type Conversion Rules

PowerShell has several mechanisms for converting between objects that do not require the conversion strategies listed in the previous section. There are well-known ways to convert one object to another, and they are built in the language. Here they are:

- Every object can be converted to a `PSObject`. In fact, the shell does so automatically, and most of the time, we are not holding real references to objects but are working with `PSObject` instances that point to the actual object.
- Conversions to void return `$null`:

```
PS C:\> ([void] 5) -eq $null
True
```

- We can convert all types of collections and objects to arrays. A collection here is defined as any .NET type that implements the `ICollection` interface. The .NET framework has many collection classes built in, and collections implemented by any programmer implement that interface too. That means we can convert to an array from any collection. Here is how:

```
PS C:\> ([object[]] 5).GetType().FullName
System.Object[]
PS C:\> $strings = New-Object Collections.Specialized.StringCollection
PS C:\> ([object[]]$strings).GetType().FullName
System.Object[]
```

- We can convert all objects to Boolean values by treating all nonnull and nonempty objects as true. We will discuss that in detail in Chapter 2.
- Anything can be converted to a string.
- `IDictionary` objects can be converted to `Hashtable`. The `IDictionary` interface in the .NET framework is implemented by objects that maintain a mapping between a set of key objects to a set of values. The `Hashtable` class is the most commonly used dictionary object, and here is how we can convert an `OrderedDictionary` object to a `Hashtable`:

```
PS C:\> $dict = New-Object Collections.Specialized.OrderedDictionary
PS C:\> $dict["item1"] = 3
PS C:\> $hash = [Hashtable] $dict
PS C:\> $hash
```

Name	Value
item1	3

```
PS C:\> $hash.GetType().FullName
System.Collections.Hashtable
PS C:\> $dict.GetType().FullName
System.Collections.Specialized.OrderedDictionary
```

- “Converting” a variable to a `PSReference` creates a reference variable pointing to the original. Modifying a reference value modifies the original:

```
PS C:\> $a = 3
PS C:\> $b = [ref] $a
PS C:\> $b.Value = 4
PS C:\> $b.Value
4
PS C:\> $a
4
```


We will discuss references in greater detail in Chapter 5, because they are particularly helpful when passing parameters to functions.

- A properly formatted string can be converted to an XML document:

```
PS C:\> $doc = [xml] "<root><item>1</item></root>"
PS C:\> $doc.root.item
1
```

- Script blocks can be converted to .NET delegates. Delegates are a special type of object that point to a method or a block of code. They are typically involved in associating event handler code with the event and are used by objects to fire an event. We usually need to convert a script block to a delegate when working with a .NET type that can fire an event that we want to consume.

A number of objects can be created or obtained by converting from a string that contains a name, path, or a query. Here they are:

- Strings can be converted to character arrays:

```
PS C:\> [char[]]"chars"
c
h
a
r
s
```

- Strings can be turned to regular expressions. The string is treated as the match pattern:

```
PS C:\> ([regex] "a.*?b").GetType().FullName
System.Text.RegularExpressions.Regex
```

- Strings can also create WMI objects (I will demonstrate the power of those conversions in Chapter 20, where we will work with WMI):

```
PS C:\> [wmisearcher]'Select * from Win32_Process'
```

```
Scope      : System.Management.ManagementScope
Query      : System.Management.ObjectQuery
Options    : System.Management.EnumerationOptions
Site       :
Container  :
```

```
PS C:\> [WMI]'\\.\root\cimv2:Win32_Process.Handle=0'
ProcessName      : System Idle Process
```

```
...
PS C:\> ([wmi] "Win32_Share").GetType().FullName
System.Management.ManagementClass
```

- Strings can be converted to Active Directory entries too. The conversion process uses the string as the `DirectoryEntry` path. Note how we use `PSBase` to get to the real object type instead of the adapted view:

```
PS C:\> $user = [ADSI]"WinNT://./Hristo,user"
PS C:\> $user.PSBase.GetType().FullName
System.DirectoryServices.DirectoryEntry
```

- Strings can point to .NET types. The input string is treated as the type name. The conversion works with the full type name. Again, we can omit the `System` namespace prefix:

```
PS C:\> [type] "System.Diagnostics.Process"
```

IsPublic	IsSerial	Name	BaseType
-----	-----		-----
True	False	Process	System.Compon...

```
PS C:\> [type] "Diagnostics.Process"
```

IsPublic	IsSerial	Name	BaseType
-----	-----		-----
True	False	Process	System.Compon...

Accessing Static Members

Type literals can be used for another important purpose: they are the only way to access a static property or call a static method. The syntax requires us to separate the type literal and the member name with a double colon (`::`). This is how we can get a static property value:

```
PS C:\> [datetime]::Now
Tuesday, September 04, 2007 9:25:38 AM
```

```
PS C:\> [datetime]::Today
Tuesday, September 04, 2007 12:00:00 AM
```

Calling static methods looks similar. In the following example, we create a `double` object by manually calling the `Parse` method:

```
PS C:\> [double]::Parse("2.5")
2.5
```

Summary

PowerShell's extended type system is really an impressive piece of work. It is very powerful and complex, and the really remarkable thing about it is that it has not turned out too restrictive or too newbie unfriendly. Users find their way around quickly because of the uniform access they get to all types of objects through type adaptation and extension. The most commonly used objects get pragmatic extensions that boost a scripter's productivity and lower the learning curve. The type conversion system silently transforms objects under the hood, adapting them as necessary, thus saving valuable keystrokes and increasing script readability. All these assets make the PowerShell language a formidable blend of a script language's simplicity with the raw power of .NET and Windows



Controlling Execution Flow

Our journey in the PowerShell programming language will now continue with a quick tour of control flow. “Control flow” is a fancy term that computer scientists and programming language pundits like to use when they mean simply telling your program what to execute next. Eventually, all control-flow-related topics boil down to discussing conditional branches, like the `if` statement, and looping or repeating an operation a number of times.

PowerShell offers a conventional approach to branching and looping by intentionally implementing constructs that are similar to the ones found in the C language family. People familiar with C, C++, C#, JavaScript or maybe even Visual Basic will feel at home, as the language tries to mimic most of the statements present in those languages. Of course, there are some differences, but they are usually offered in the form of enhancements. The basic C-inspired syntax will work and not knowing the advanced forms of these operators will not stop anyone from doing a fine job at scripting. But familiarity with one of these languages is not required: we will start with the basic syntax. Learning the PowerShell control flow constructs will bring an additional benefit to the newbie scripter—once you understand PowerShell control flow, you will be able to read and understand the tons of automation examples written in VBScript or JScript that can be found on the Web.

Conditional Expressions

Before you learn to branch your program execution according to some condition, you must learn about conditional expressions. They are the means of checking if a condition is true, and they are the primary tool used in statements like `if`, `switch`, `for`, and `while`, which we will look into later in this chapter.

So, what is a conditional expression? Just like an arithmetic expression, a conditional one is a set of operations that are evaluated in order to compute a result. However, conditional expressions always evaluate to Boolean values, that is, they always return `$true` or `$false`. For example, this is how to check if the variable `$num` contains a number that is less than five:

```
PS> $num = 4
PS> $num -lt 5
True
```

For now, treat the `-lt` operator as the way to check if the left-hand value is less than the right-hand one. I will get to details about this and the other comparison operators in a minute.

A conditional expression is, after all, an expression, so we can include property references and method calls:

```
PS> $user = @{"Name"="John"; "Age"=30}
PS> $user.Age -lt 20
False
PS> $user.ContainsValue("John")
True
```

We can even include calls to PowerShell functions, cmdlets, and script blocks. For example, we can start a `notepad.exe` process and then check if the number of running `notepad.exe` processes is less than two:

```
PS> (Get-Process notepad).Length -lt 2
True
PS> notepad.exe
PS> (Get-Process notepad).Length -lt 2
False
```

The preceding code uses the `Get-Process` command to obtain all `notepad.exe` processes. You can learn more about getting process details and managing processes in Chapter 14.

Comparing Values

Most programming languages use mathematical symbols to do value comparisons: `<`, `>`, `<=`, `>=`, `=`. PowerShell is different in that instance. It tries to be first a shell and second a programming language. All shells have the `>`, `>>`, and `<` operators reserved for command output and input redirection and so does PowerShell. That made the language designers choose to use switches as comparison operators. Switches look like the abbreviated operator names:

- `-eq`: The equal switch will return true if two objects are equal, otherwise, false:

```
PS> 3 -eq 3
True
PS> 3 -eq 4
False
```

- `-ne`: The not-equal switch is the inverse of the equality operator, and it exists primarily for convenience so that we do not have to wrap equality tests in negation operations:

```
PS> 3 -ne 3
False
PS> 3 -ne 4
True
```

- `-lt`: This less-than switch will return true if the left-hand operand is less than the right-hand one:

```
PS> 3 -lt 4
True
```

- `-gt`: The greater-than switch is the inverse of the less-than one; it will return true if the left-hand operand has a greater value than the right-hand one:

```
PS> 5 -gt 4
True
```

- `-le`: The less-than-or-equal switch tests if the first value is less than or equal to the second one:

```
PS> 3 -le 4
True
PS> 3 -le 3
True
```

- `-ge`: The greater-than-or-equal switch checks if the first value is greater than or equal to the second one:

```
PS> $a = 4
PS> $a -ge 4
True
PS> $a--
PS> $a -ge 4
False
```

The `$a` statement in the previous example is a variable reference. The variable gets assigned the value 4 and is then compared to see if it is greater than or equal to four.

Strings can be compared too, so we can see if one is greater than another. The operation is defined in terms of the alphabetical order, meaning strings are compared character by character. If two strings start with the same character sequence, the shorter one is picked as the one of smaller value. Here are several comparisons that illustrate the rules:

```
PS> "a" -lt "b"
True
PS> "a" -lt "ba"
True
PS> "aa" -lt "aaa"
True
```

A thing to keep in mind is that, by default, character casing is ignored when comparing strings. That means that “John” equals “JOHN”:

```
PS> "John" -eq "JOHN"
True
```

This is a good default, as most of the time, we do not need case-sensitive comparisons. If we do need them, we have to use the case-sensitive versions of the comparison operators: `-ceq`, `-clt`, `-cle`, `-cgt`, and `-cge`. Here is how we can compare two strings that differ in their casing only:

```
PS> "john" -lt "JOHN"
False
```

The previous example returns false, because when doing a case-insensitive comparison, the two strings are considered equal. Using the case-sensitive operator tells us that the lower-case version is less than the uppercase one:

```
PS> "john" -clt "JOHN"
True
```

We can verify this by changing the operator:

```
PS> "john" -cgt "JOHN"  
False
```

The case-sensitive greater-than operator returned `$false`, so “john” must truly be less than “JOHN”. For the sake of completeness, let’s see that the two strings are not equal:

```
PS> "john" -ceq "JOHN"  
False
```

PowerShell has explicit case-insensitive operators too. To use those, we use the prefix `i` much in the same way that we used `c` in the case-sensitive ones: `-ieq`, `-ilt`, `-ile`, `-igt`, and `-ige`. Why do we need this? What is the difference between the default comparison operators and the case-insensitive ones? In other words, how does `-eq` differ than `-ieq`? The answer is simple: there is absolutely no difference. Both the default and the case-insensitive operators do the same thing. The value in using the case-insensitive versions of the operators is in increasing script readability. Seeing a case insensitive operator in a string comparison clearly conveys the intention of the original code author and makes code easier to understand.

Implicit Type Conversions

Remember that we can explicitly convert values from one type to another? PowerShell tries to be helpful and will do many conversions automatically. The general rule is that an expression that has two variables of different types will be evaluated after attempting to automatically convert the right-hand value to the type of the left-hand object. To illustrate this, here is how we can compare a string to a number:

```
PS> 2 -eq "2"  
True
```

The “2” string got converted to the number 2. The inverse conversion will also work:

```
PS> "2" -eq 2  
True
```

Remember the rule: implicit conversions always convert the right-hand value. That may yield seemingly illogical results at first; for example, two variables may be equal or not equal depending on the order of comparison:

```
PS> 2 -eq "02"  
True  
PS> "02" -eq 2  
False
```

The first expression converts the “02” string into the number 2. That makes the two values equal. The second one converts the number 2 into the string “2”. Of course, that is different than “02”. The same model can be applied to the rest of the comparison operators:

```
PS> 5 -lt "04"  
False  
PS> "05" -lt 4  
True
```


Here, the number 5 is not less than 4 (the number that the “04” string is converted to). In the second expression, alphabetically the string “05” comes before the string “4”, so the less-than operation returns true.

Logic and Bitwise Operations

Quite often, we need to combine more than one condition into a complex expression. We do that by using the logic operators. PowerShell supports the standard operations present in every language:

-and

The -and operator returns \$true if both operands evaluate to \$true. Here is how we can use it to check if a file has a .txt extension and its size is greater than 10KB at the same time:

```
PS> dir largertext.txt
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\PowerShell
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	9/13/2007 6:33 AM	11266	largertext.txt

```
PS> $file = Get-Item largertext.txt
PS> $file.Extension -eq ".txt" -and $file.Length -gt 10KB
True
```

Note that having any one of the conditions return \$false will yield \$false for the entire expression:

```
PS> $file = Get-Item smalltext.txt
PS> $file.Extension -eq ".txt"
True
PS> $file.Length -gt 10KB
False
PS> $file.Extension -eq ".txt" -and $file.Length -gt 10KB
False
```

-or

The -or operator returns \$true if any of its operands evaluates to \$true. Here is how we can use it to check if a file has been created or modified today:

```
PS> $file.CreationTime.Date -eq [datetime]::Today -or `
    $file.LastAccessTime.Date -eq [datetime]::Today
True
```

We are using the `CreationTime` and `LastAccessTime` properties to get the actual times of creation and access. To truncate the time part and use only the date, we get the `DateTime.Date` property.

-xor

`-xor` is the exclusive or operator. It returns `$true` if one of the operands is `$true`, but it will return `false` if both of them are `$true`. We can use it, for example, to check if a file has a `.txt` extension or is read only, but not both:

```
PS> attrib +r test.txt
PS> attrib -r test2.txt
PS> $file1 = Get-Item test.txt
PS> $file1.Extension -eq ".txt" -xor $file1.IsReadOnly
False
PS> $file2 = Get-Item test2.txt
PS> $file2.Extension -eq ".txt" -xor $file2.IsReadOnly
True
```

Note that we set the read-only attribute to `test.txt` and clear it for `test2.txt` before evaluating our `-xor` expressions.

-not

The `-not`, or `!`, operator is different in that it works with only one operand. All it does is return the negative of its operand: if the input is `$true`, it will return `$false`, and vice versa. Note that this operator has two forms: `-not`, the switch-based one, and the `!` prefix. The latter is again a tribute to the C language family, where all languages use the exclamation point as a negation operator. Using one form over the other is a matter of preference. Let's now check that a file is *not* read only:

```
PS> $file = Get-Item test2.txt
PS> $file.IsReadOnly
False
PS> -not $file.IsReadOnly
True
PS> !$file.IsReadOnly
True
```

Note that even the switch form of the operator has to be placed before the operand.

PowerShell includes a bitwise version of the `-and` and `-or` operations: the `-band` and `-bor` operators. The bitwise operations work on integers only. They go over each bit in one of the numbers and compare it with the corresponding bit in the other number. The result is an integer that has its bits set in those positions where the bit operations returned `$true`. Although PowerShell does not support binary number literals, binary operations are best understood if we treat all the numeric input as binary. Here is how we can use `-band` to and all the bits of the numbers 3 and 1:

```
PS> 3 -band 1
1
```

This needs some explanation. The number 3 is represented by the binary number 11. 1 is represented as 01. `-band` goes over each of the bits and, in our example, sets the rightmost bit only, as the rightmost position is the only one that contains two ones.

```

      11
-band 01
=     01

```

Now, let's look at how `-bor` works:

```

PS> 2 -bor 1
3

```

The number 2 is 10 in binary. Again, 1 is 01. The binary or operation will yield 11 or a decimal 3.

```

      10
-bor  01
=     11

```

Why bother with binary numbers and bitwise operations? The truth is that bitwise operations are pretty rare in scripting languages. You will need them only when you interface with the outside world. For example, you might have to talk with a program that returns result flags as a number, and to correctly test that a flag is set, you have to use a bitwise and operation. The most common case when you will need that is when you interact with a .NET object that will return a `Flags` enumeration. Those enumerations' numeric values are carefully chosen so that they occupy different bits, and they can be combined using binary or operations so that a single number can contain several values. The first example of such an enumeration that comes to mind is a file attribute. A `FileInfo` object's `Attributes` property will return a `System.IO.FileAttributes` enumeration whose values are as shown in Table 2-1.

Table 2-1. *System.IO.FileAttributes Values in Hexadecimal*

Name	Value
ReadOnly	0x01
Hidden	0x02
System	0x04
Archive	0x20
Directory	0x10
Device	0x40
Normal	0x80
Temporary	0x100
SparseFile	0x200
ReparsePoint	0x400
Compressed	0x800

Table 2-1. *System.IO.FileAttributes Values in Hexadecimal (Continued)*

Name	Value
Offline	0x1000
NotContentIndexed	0x2000
Encrypted	0x4000

Hexadecimal is very convenient when working with binary values because a single hex digit represents four bits, or numbers from 0 to 15. Take the `ReadOnly`, `Hidden`, and `System` values for example. Their values in binary are

```
ReadOnly: 0001
Hidden:   0010
System:   0100
```

When you get a value of 3 for the rightmost hex digit, you know that the file is both `ReadOnly` and `Hidden`, because 1 -or 2 equals 3 or 11 in binary. Now, let's use this to test if a file is read only:

```
PS> $file = Get-Item c:\pagefile.sys -force
PS> $file.Attributes -band 0x01
0
PS> $file.Attributes -band 0x02
2
PS> $file.Attributes -band 0x04
4
```

The first operation returns 0; that means that the file is not read only. The next two operations return nonzero values; that means that the corresponding bits are set; in other words, the file is both hidden and system. Note that we had to use the `-force` switch when calling `Get-Item` to get the hidden file.

To test that a file is both hidden and system, you can `-band` its attributes with the value 6, the result of 2 `-bor` 4:

```
PS> $file.Attributes -band 0x06
6
```

It is always best to choose readability over compactness, so the preceding example is better written as:

```
PS> $file.Attributes -band (0x02 -bor 0x04)
6
```

Or even better, we can introduce variables that explain the intent:

```
PS> $hidden = 0x02
PS> $system = 0x04
PS> $file.Attributes -band ($hidden -bor $system)
6
```

Boolean Conversions

The bitwise operators and many more operations do not return a Boolean value. They return a number. How do we convert those values to `$true` or `$false`? No problem! PowerShell does that automatically. The short version of the rules for Boolean conversions is that any nonempty value will be converted to `$true`. The “nonempty” concept can be stretched quite a lot, so let’s define it. PowerShell will implicitly convert to a Boolean value when needed, but we will use an explicit conversion to force a conversion and better see the results. We can force the conversion by using the `[bool]` type literal before any value. Here are the conversion rules:

- Any number that has a value different than 0 will be converted to `$true`:

```
PS> [bool] 1
True
PS> [bool] 5
True
PS> [bool] 5000
True
PS> [bool] 0
False
```

That means you can use numbers in conditional expressions, as they will be converted for you:

```
PS> $true -and 5000
True
```

- Strings that have a nonzero length will be converted to `$true`:

```
PS> [bool] "true"
True
PS> [bool] "false"
True
PS> [bool] "something else"
True
PS> [bool] ""
False
```

- Collections that have at least one item are equal to `$true`:

```
PS> [bool] (1,2)
True
PS> [bool] ("one", "two")
True
PS> [bool] @()
False
```

- Other objects are converted to `$true` unless they are `$null`. Assuming that Microsoft Word is running, here is how its object gets converted to a Boolean value:

```
PS> [bool] (Get-Process winword)
True
```

- The same holds true for any object—only `$null` values ever get converted to false:

```
PS> $user = @{Name="John"; "Age" = 30}
PS> [bool] $user
True
PS> [bool] $null
False
```

The code in the preceding examples uses explicit type casts just to illustrate what PowerShell does under the hood.

The implicit conversions can be used to do a really compact conversion to Boolean by using a clever negation trick—double negation:

```
PS> !!2
True
PS> !!"test"
True
PS> !!0
False
PS> !!$null
False
PS> !!(Get-Process winword)
True
```

See the two exclamation marks? The first one converts the original value to Boolean and then negates it. The second one negates again and, in effect, cancels the previous negation, returning the original value converted to Boolean. I would like to stress again the importance of readability over compactness: use the double negation trick in quick tests on the command line, but prefer explicit casts in larger scripts, so that the intent of the code becomes immediately clear.

The Power of String Conversions

We already discussed the `-like` and `-match` string operators in Chapter 1. While it is true that we can use them to test if a string is composed according to a specific pattern or is of a form that we want to detect, it is really useful when you combine that with converting various objects to strings. This way, we can make our conditional expressions much more compact and expressive. Let's get back to our previous example that checked file attributes. The `System.IO.FileAttributes` enumeration can be easily converted to a string. When we do that, the string will contain the names of the set bits. This is how that enumeration looks for a typical `pagefile.sys` file:

```
PS> $file = Get-Item C:\pagefile.sys -force
PS> $file.Attributes
Hidden, System, Archive
PS> [string] $file.Attributes
Hidden, System, Archive
```

We do not need the explicit cast—PowerShell will do that automatically:

```
PS> $file.Attributes
Hidden, System, Archive
```

Now, we can use the `-like` operator to test if the file is hidden or system:

```
PS> $file.Attributes -like "*Hidden*"
True
PS> $file.Attributes -like "*System*"
True
```

What if we want to test if the file is hidden or system? We can use `-or`:

```
PS> $file.Attributes -like "*Hidden*" -or $file.Attributes -like "*System*"
True
```

This is too wordy. We can use a regular expression with an alternation pattern to test both things at the same time. We have to switch to the `-match` operator:

```
PS> $file.Attributes -match "Hidden|System"
True
```

The expression is equivalent to the previous one. We have just moved the `-or` operation to the alternation operator inside the regular expression.

Okay, which operator should we use when working with flag enumerations: bitwise operators or string matches? There is really no right answer, and it all boils down to matters of preference. Bitwise operations feel a bit more robust, as it is extremely unlikely that future versions of the framework will change the enumeration values. Checking an enumeration value using bitwise operation is the better solution performance-wise, as it will be faster than the string operators. The string version is robust enough, though—the enumeration string values are not likely to change either. Using strings gets us a more compact solution, and some people may argue that it is more readable too.

Collections and Conditional Expressions

Most programming languages treat collections as single objects. They allow us to write conditional expressions by checking properties of the collection object or by using one of the objects contained inside the collection. PowerShell adds an interesting twist to this. It allows us to use a collection in the left-hand side of a conditional expression. When the shell sees that, it applies the conditional expression to each of the members of the collection. The result is a new collection that contains the members that have been evaluated to `$true` using that operation. In effect, this will filter the collection. Here is how we can use that:

```
PS> 1,2,3,4 -lt 3
1
2
PS> 1..5 -le 4
1
2
3
4
```

We can use all conditional operators with collections containing any type of objects. Here is how to get only strings equal to “John”:

```
PS> "John", "Mike", "John", "Jane" -eq "John"
John
John
```

Of course, all implicit conversions remain in effect:

```
PS> 3,4,5 -eq "4"
4
```

Branching Instructions

Branching instructions bear that name, because they split program execution according to a conditional statement. Typically they specify several possible execution flows. At the time of execution, a statement is evaluated, and a code path is chosen according to the value returned from that statement. PowerShell offers two branching instructions: `if/else` that allows us to choose one of two code paths, and `switch` that selects and executes one of many alternatives.

Simple Branching: `if/else`

The simplest branching instruction in many mainstream programming languages is the `if` statement. It evaluates a conditional expression and executes one block of code if the result is true, another if false. The general form of the statement is

```
if (<conditional>)
{
    <true code block>
}
else
{
    <false code block>
}
```

The `else` part is optional. Omitting it will execute nothing if the conditional expression evaluates to `$false`. Here is an example that displays a different message if the file size is greater than 10KB:

```
PS> $file = Get-Item largertext.txt
PS> if ($file.Length -gt 10KB){
    Write-Host "More than 10 kilobytes"
}
else{
    Write-Host "Less than 10 kilobytes"
}
```

More than 10 kilobytes

`if` statements can be used for more complex branching if we nest several of them. This is how we can display a different message if a file is larger than 10KB, between 10KB and 5KB, and less than 5KB:


```
PS> $file = Get-Item smalltext.txt
PS> if ($file.Length -gt 10KB){
    Write-Host "More than 10 kilobytes"
}
else{
    if ($file.Length -lt 5KB){
        Write-Host "Less than 5 kilobytes"
    }
    else{
        Write-Host "Between 5 and 10 kilobytes"
    }
}
```

Between 5 and 10 kilobytes

Nesting if statements quickly becomes unbearable, as it adds more noise than real value, and that is why PowerShell provides a shortcut—the `elseif` optional part to the `if` statement. It can be used to evaluate another condition, and if true, execute its associated branch. If all conditions evaluate to `$false`, the `else` part gets evaluated. Here is how we can rewrite the above using `elseif`:

```
PS> $file = Get-Item smalltext.txt
PS> if ($file.Length -gt 10KB){
    Write-Host "More than 10 kilobytes"
}
elseif ($file.Length -lt 5KB){
    Write-Host "Less than 5 kilobytes"
}
else{
    Write-Host "Between 5 and 10 kilobytes"
}
```

Between 5 and 10 kilobytes

Multiple Branches and the switch Statement

Like many other PowerShell language features, the `switch` statement comes from the C language. The C version takes a number and executes a different action according to the number value. PowerShell takes that a bit further and allows using all types of objects. The general form of a `switch` statement looks like this:

```
switch (<variable>)
{
    value1 { <action1> }
    value2 { <action2> }
    ...
    default { <default action> }
}
```

See the mapping between expected values and the respective actions? The default part is executed if the actual value matches none of the expected ones. Here is how to get the digit name from a number using a switch statement:

```
PS> switch (3){
    1 { "one" }
    2 { "two" }
    3 { "three" }
    default { "unknown digit" }
}
```

three

```
PS> switch (7){
    1 { "one" }
    2 { "two" }
    3 { "three" }
    default { "unknown digit" }
}
```

unknown digit

An interesting peculiarity of the switch statement is that it will evaluate all blocks whose value matches:

```
PS> switch (1){
    1 { "digit" }
    "1" { "string" }
}
```

digit

string

To stop evaluation after the current block, you have to use the break statement, which will exit the switch evaluation:

```
PS> switch (1){
    1 { "digit"; break }
    "1" { "string" }
}
```

digit

You're probably already guessing how switch compares values during execution. That's right: by default, it uses the -eq operator. That means type conversions are done implicitly, and string comparisons will be case insensitive. To force using a case-sensitive comparison or using the -ceq operator, you have to pass the -casesensitive switch:

```
PS> switch ("john"){
    "john" { Write-Host "lowercase" }
    "JOHN" { Write-Host "uppercase" }
}
```

```

lowercase
uppercase
PS> switch -casesensitive ("john") {
    "john" { Write-Host "lowercase" }
    "JOHN" { Write-Host "uppercase" }
}

```

```
lowercase
```

`-casesensitive` is not the only operator that `switch` can use. It also supports the `-wildcard` and `-regex` switches that will match a value using the `-like` and `-match` operators respectively:

```

PS> switch -wildcard ("johnson"){
    "john*" { Write-Host "starts with john" }
    "j*son" { Write-Host "starts with j, ends with son" }
}

```

```
starts with john
```

```
starts with j, ends with son
```

Here is how to do a similar thing with regular expressions:

```

PS> switch -regex ("johnson"){
    "(john.*)|(.son)" { Write-Host "starts with john or ends with son" }
}

```

```
starts with john or ends with son
```

Combining `-wildcard` or `-regex` with `-casesensitive` will make `switch` use the `-clike` and `-cmatch` operators, respectively, to do case-insensitive comparisons:

```

PS> switch -regex -casesensitive ("JOHNSON"){
    "(john.*)|(.son)" { "Starts with john or ends with son" }
    default { "No match" }
}

```

```
No match
```

The `switch` statement can work for collections too. It goes over all members of the collection, executes the script block that matches the member value, and returns a collection that contains the return values of all switch operations. Going back to our example that converts a number to a string with the digit name, here is how we can transform multiple digits:

```

PS> switch (3, 2, 1, 7){
    1 { "one" }
    2 { "two" }
    3 { "three" }
    default { "unknown digit" }
}

```

```
three
two
one
unknown digit
```

The input collection can come from a file too. Passing the `-file` switch will get the file contents and will cause PowerShell to treat it as a collection, using each line as a member. The example below writes four digits to a file and then uses a `switch` statement to read them back and return a different value for each digit:

```
PS> (3,2,1,7) | Set-Content digits.txt
PS> switch -file digits.txt{
    1 { "one" }
    2 { "two" }
    3 { "three" }
    default { "unknown digit" }
}
```

```
three
two
one
unknown digit
```

Loops: Repeatedly Executing Blocks of Code

Loops are the last piece of the control flow puzzle. They allow us to repeatedly execute a block of code, usually while a condition is satisfied. We can use them to repeat an action many times, to walk over a collection of objects and perform an action on each and every one of them, wait indefinitely for an external condition to become true, and much more.

The Simplest Loop: `while`

What could be simpler than the ability to repeat something while a condition is true? The `while` loop allows us to do exactly that. Its general form is

```
while (<condition>)
{
    <action block>
}
```

Here is how to use it to wait for the `notepad` process to exit:

```
PS> while (Get-Process notepad -ErrorAction SilentlyContinue){
    Write-Host "Waiting for notepad to exit"
    sleep 1
}
```

```
Waiting for notepad to exit
Waiting for notepad to exit
Waiting for notepad to exit
```

The `Get-Process notepad -ErrorAction SilentlyContinue` statement is executed in every iteration, and it is converted to a Boolean value. The `Get-Process` cmdlet will raise an error if it cannot find a process, so we pass the `-ErrorAction SilentlyContinue` parameter to suppress the error and just get a `$null` value back if no process is found; the `$null` value will be converted to `$false`, and the loop will exit. We do not want to hog the entire CPU time in useless looping while waiting for the process to exit, so we suspend execution for one second at the end of each loop iteration by calling the `sleep` command.

The `while` loop body will never execute if the condition initially evaluates to `$false`. To have a loop that is guaranteed to execute its body at least once, we have to use the `do-while` form:

```
PS> $i = 5;
PS> do{
    Write-Host 'incrementing $i'
    $i++
}while ($i -le 1)
```

```
incrementing $i
```

Note that the body executed once even with the condition being false at the time we entered the loop body.

Sometimes, it feels awkward to artificially negate a condition to loop until a condition becomes true. PowerShell provides a `do-until` loop that helps us in such cases:

```
PS> $i = 0;
PS> do{
    Write-Host "`$i = $i"
    $i++
}until ($i -ge 3)
```

```
$i = 0
$i = 1
$i = 2
```

The `do-until` loop behaves in the same way as `do-while` except that the condition is negated. Here is the same loop written using `do-while` with a negated condition:

```
PS> $i = 0;
PS> do{
    Write-Host "`$i = $i"
    $i++
}while (-not ($i -ge 3))
```

```
$i = 0
$i = 1
$i = 2
```

Loops and Counters: The for Loop

As you have seen from the previous examples, many loops involve initializing a counter variable that gets modified in every loop iteration and is, in turn, used in a conditional expression that serves as the loop exit condition. This pattern repeats quite often, and that is why the PowerShell language has the for loop construct. Its general form is

```
for (<initializer>; <exit condition>; <step action>)  
{  
    <action>  
}
```

The loop is prepared by executing the initializer action. Usually, this involves initializing a counter variable. Every loop iteration executes the step action, which typically increments or decrements the counter variable. Every iteration is executed if and only if the exit condition returns true when it's evaluated. Here is how to write the while loops you saw before that iterate from zero to two by using a for loop:

```
PS> for ($i = 0; $i -lt 3; $i++){  
    Write-Host "`$i = $i"  
}  
  
$i = 0  
$i = 1  
$i = 2
```

That is much more compact than the while version. Now that I've mentioned while, you can see that the for loop has a superset of while's functionality. In other words, while loops can be emulated using for loops. The initializer and step portions of a for loop are optional, and omitting them gets us with an equivalent of while:

```
PS> $i = 0;  
PS> for (;$i -lt 3;){  
    Write-Host "`$i = $i"  
    $i++  
}  
  
$i = 0  
$i = 1  
$i = 2
```

Note that we still need the semicolons to indicate that the initializer and step expressions are empty and the expression passed inside the parentheses should be used as the loop exit condition.

Executing Actions on All Objects in a Collection: The foreach Loop

Quite often, we have to write code that walks over a collection and acts on each of its members. Here is how we can do that with a for loop:

```
PS> $items = 2,3,4
PS> for ($i = 0; $i -lt $items.Length; $i++){
    $item = $items[$i]
    Write-Host $item
}

2
3
4
```

Some parts of the loop will always be the same no matter what collection we are using: the `$i` variable that contains the index of the current item, the `$i -lt $items.Length` exit condition, the increment statement used as a loop step, and the `$item` variable initialization with the current item. Realizing that, the PowerShell language designers have added another looping facility: the `foreach` loop. The general form looks like this:

```
foreach ($item in $collection)
{
    <action>
}
```

Note that `foreach` requires us to name the variable that contains the current item and provide the input collection. That is it! The index initialization, the exit condition, and the index increment are all hidden under the hood. Here is how we can use `foreach` to better write the previous loop:

```
PS> $items = 2,3,4
PS> foreach ($item in $items){
    Write-Host $item
}

2
3
4
```

Interestingly enough, `foreach` is a reserved word in PowerShell, and at the same time, it is used as an alias for the `ForEach-Object` cmdlet. The good news is that the PowerShell language parser is smart enough to detect when you are using a `foreach` loop and when you are creating a pipeline that uses `ForEach-Object`. I will cover object pipelines and the `ForEach-Object` cmdlet in the next chapter.

Controlling Loop Execution: `break` and `continue` Statements

Loops do not necessarily have hard exit conditions. We might wish to force exiting a loop if we detect something bad or just wish to optimize our program and avoid excessive iteration. We can do that by using the `break` statement. Here is how to use it to get the first file that has a `.log` extension in the current folder:

```
PS> foreach ($file in dir){  
    if ($file.Name -like "*.log"){  
        Write-Host $file.Name  
        break  
    }  
}
```

kill.log

Since we need the first file only, there is no point in continuing iterating over the collection once we find it. That is why we exit the loop using the break statement.

Quite often, we want to skip executing the rest of the loop body if a condition is true. The most common case is when we iterate over a collection, perform actions on most of the items, and wish to skip some of the items. We can do that using the continue statement. It behaves in a way similar to break, with the only difference that it does not exit the loop—it exits the current iteration only and execution skips to the next iteration. Here is how we can use continue to walk over a collection writing all odd numbers to the console and skipping the even ones:

```
PS> foreach ($item in 1..5){  
    if ($item % 2 -eq 0){  
        continue  
    }  
    Write-Host $item  
}
```

1
3
5

We are using the modulo operator, %, to check if a number is even or odd. Even numbers are divisible by two and yield zero as their modulo two value.

break and continue have some advanced uses: they can be given a loop label, which will instruct them to exit or go to the next iteration for the loop in question. This is useful when you have nested loops and wish to control the outer loop from the inner loop body. Here is an example of two nested foreach loops:

```
PS> foreach ($outerItem in 1..3){  
    Write-Host "outerItem: $outerItem"  
  
    foreach ($innerItem in 1..3){  
        Write-Host "innerItem: $innerItem"  
        break;  
    }  
}
```



```
outerItem: 1
innerItem: 1
outerItem: 2
innerItem: 1
outerItem: 3
innerItem: 1
```

Note how the inner loop iterates only once for each outer loop iteration. That is because the `break` statement terminates the loop after the first `Write-Host` statement. What if we want to exit the outer loop instead of the inner one? We have to label the outer loop and use the label when executing `break`:

```
PS> :outerLoop foreach ($outerItem in 1..3){
    Write-Host "outerItem: $outerItem"

    foreach ($innerItem in 1..3){
        Write-Host "innerItem: $innerItem"
        break outerLoop;
    }
}
```

```
outerItem: 1
innerItem: 1
```

This time both the outer and inner loops iterate only once. The inner `break` statement exits the outer loop, effectively terminating execution.

We can use `continue` with labels too. Here is what happens when we continue the next outer loop iteration from within the inner loop:

```
PS> :outerLoop foreach ($outerItem in 1..3){
    Write-Host "outerItem: $outerItem"

    foreach ($innerItem in 1..3){
        Write-Host "innerItem: $innerItem"
        continue outerLoop;
    }
}
```

```
outerItem: 1
innerItem: 1
outerItem: 2
innerItem: 1
outerItem: 3
innerItem: 1
```

The result is the same as with our first example, because continuing the outer loop effectively breaks out of the inner one.

Summary

This chapter covered the second part of the PowerShell essentials. Armed with the knowledge about the type system that you acquired in Chapter 1 and knowing how to control your program execution by using conditional expressions in branches and loops, you are now ready to start writing fully functional commands that help you in your daily activities.

The next chapter will raise the level of abstraction with regard to looping over collections of objects and will introduce you to object pipelines. Taking the time to practice working with loops, branches, and conditional expressions before moving on may be worthwhile, because mastering these topics is the key to understanding the automatic operations that happen when we compose pipelines.



The Object Pipeline

The primary use for a shell is to run commands. The shell interprets user input and launches one or more commands according to that. A user who wants to specify a complex action to perform has to invoke the right commands in the right order with the right data. Because creating a complex command that does many things and is flexible enough to meet everyone's needs is difficult, most shells move away from monolithic do-it-all commands and instead concentrate on providing small tools that can combine their power into something big. Each of those tools usually does one specific job extremely well. For example, a hypothetical command that searches through all files with the .txt extension in a project folder and displays all occurrences of the "financial report" phrase is hard to make very flexible. What if we want to search in files modified last week? It is much easier to create two commands: Find-File to find files according to criteria such as file extension, modification date, and so on and Search-Contents to search for words or phrases inside files.

The biggest benefit of separating the commands is that, this way, they can be used independently and in conjunction with other commands. This opens up a lot of possibilities for new uses that the original developers may not have even imagined. Bigger and more complex functionality emerges from those uses. The driving force behind that process, and its primary enabler, is an intuitive and simple mechanism for composing commands and passing data among them. The most powerful way to do pass data is to chain commands into pipelines: pipelines allow us to send one command's output as a second command's input. Having pipelines in place means that the simplest implementation for our example search command is to send Find-File's output, a list of file names, directly to Search-Text and to perform the a text search on the found files.

Text-Based Pipelines

The pattern of chaining commands has been in practice for quite some time, and it has been implemented in most shells like so: each command reads text as its input and, in turn, generates text as its output. This is very easy to implement and provides a basic way to pass data around. Unfortunately, using text as the data transport has several drawbacks. The most important one is that there is no standard data format. Each command is free to generate whatever text it wishes, and that makes consuming data hard, as we have to parse the input and make something meaningful out of it. It is considered good form among command developers to generate output that is easy to parse, as that makes a tool more usable. Of course, output that is easiest for a machine to parse is not always in the best form for humans to read. To avoid having to make a trade-off of having either a machine-parsable or human-readable output,

developers provide switches and command-line options that control the output format according to the data consumer.

Most shells solve the parsing problem by providing helper commands that can extract data from text by applying common operations. Some of those tools follow:

- `grep` or `findstr` searches for lines that match a regular expression.
- `sed` allows users to edit the stream of text, such as replacing certain strings with something else.
- `awk` treats text as a table and splits cells according to a separator, such as a comma or tab.

Using commands like the previous ones may solve the problem but tends to create fragile solutions, as the commands may not always format text the way you expect. Handling error conditions is the most obvious example. Few text-based programs return a proper error code when something bad happens, and often, you are left parsing the returned text to figure out what went wrong. This approach is extremely brittle, as you have to be aware of all error messages and be able to parse each one of them. Even if you succeed, your code may fail in multilingual environments. A script that, for example, looks for messages of the sort “Error type 3” might get a “Fehlerart 3” message when run on a machine localized in German. Of course, we should not forget the possibility that a different parameter or a new version of the command may easily break your script by changing the format of the emitted error text. Many software developers try to keep backward compatibility with new releases of their products, but the format of the text messages is usually considered safe to change.

Another major inconvenience is that you have to learn regular expressions for use with tools like `grep`. Regular expressions are a separate language and particularly hard to learn. The `sed` and `awk` commands each have their own languages too. You certainly can do your job using just text, but you’d end up creating complex pipelines that look like Klingon to the uninitiated. The really sad part, though, is that you’d spend most of your time manipulating text instead of concentrating on the task at hand.

Object-Based Pipelines

Now that you’ve seen some of the serious limitations in passing text between commands, we can focus on a solution that will eliminate the need to do so using PowerShell’s approach to pipelining. PowerShell postulates that each command—in PowerShell parlance, `cmdlet`—will be a .NET object that satisfies a contract and has a standard interface. Building on .NET allows the shell to pass .NET objects instead of text. This thing is huge! It means that we no longer have to generate parsable text, nor do we have to parse text coming from another program. The pipeline is not a text stream; it is a full-blown collection containing real, live objects. Those objects expose a set of properties and methods that already contain any necessary information in an easy-to-access form. There’s no need to get the value from the third column of a text table that uses commas as a separator when you can just get the `Name` property.

Having objects at your disposal makes commands such as `grep`, `awk`, and `sed` irrelevant. Yet, we need analogues that work well with objects. We want to filter a collection according to some criteria, get a subset of an object’s properties, and even add our own properties. Most of the time, we also want to treat a collection of objects like a database table and query it to extract information. PowerShell introduces several `cmdlets` that help us with that task:

- `ForEach-Object`, or just `foreach`, allows you to execute an action on each item in the pipeline. The result is a new collection that contains the return values from all actions. Here is an example that multiplies all numbers in a collection by 2 and returns a new collection with the results:

```
PS> 1,2,3 | ForEach-Object {$_ * 2}
2
4
6
```

- `Where-Object`, aliased as `where` or just `?`, will filter a collection and return only items that match a condition. The name comes from the databases world, where the SQL clause that filters a data table is called `WHERE`. Let's use it to get all strings that start with "Error":

```
PS> "Error 2","John","Mike","Error 5" | Where-Object {$_ -like "Error*"}
Error 2
Error 5
```

The preceding two operations are the essentials we need for manipulating our objects. Note that we only need two operations and not the text-based commands to extract numbers or quoted strings in the first place. All that because our input has already been parsed by the shell into a collection of objects!

Besides the essentials, PowerShell comes with several additional cmdlets that make life even easier for us:

- `Select-Object`, aliased as `select`, will create a collection of objects that contain a subset of properties of the original objects' properties. Again, the name comes from the SQL language: `SELECT` is the operator that defines which table columns will be retrieved. Suppose we are interested in the physical memory a process occupies. Here is how to get it by displaying just the `ProcessName` and `WS` (working set) properties for the `WINWORD` process:

```
PS> Get-Process winword | Select-Object ProcessName,WS

ProcessName                               WS
-----
WINWORD                                     47812608
```

- `Sort-Object`, also known as `sort`, is a handy operation that sorts a collection on one or more properties. Apart from basic sorting, it also supports advanced scenarios: want to use the French culture string comparison rules for your sorts? No problem! Here is an example command:

```
PS> "cote", "côte", "coté", "côté" | Sort-Object -Culture "fr-FR"
cote
côte
coté
côté
```

- Tee-Object, shortened to tee and named after the letter “T”, will forward what it has been given to the next command and store the current pipeline into a file or variable. The command is an excellent tool for preserving a collection at some stage in the pipeline before passing it to the rest of the commands. Here is how we can save our original unsorted collection in the \$unsorted variable just before sorting:

```
PS> 3,2,1 | Tee-Object -variable unsorted | Sort-Object
1
2
3
PS> $unsorted
3
2
1
```

- Group-Object, or group, splits the objects into several groups according to a property value. Want to group a number of files according to their extensions? Here is how:

```
PS> dir | Group-Object -property Extension

Count Name                               Group
-----
      1 .log                             {kill.log}
      4 .txt                             {test.txt, test2.txt, test3.txt, test4.txt}
```

- Measure-Object calculates statistics about the collection, providing an easy way to get the minimum, maximum, average, and sum of a property value. The command is ideal for displaying quick statistics about an array of numbers:

```
PS> 1,2,3 | Measure-Object -Sum -Max -Min -Average

Count      : 3
Average    : 2
Sum        : 6
Maximum    : 3
Minimum    : 1
Property   :
```

- Compare-Object, aliased as diff, compares two objects or collections and reports on their differences:

```
PS> diff (1,2,3) (1,2,4)

InputObject SideIndicator
-----
          4 =>
          3 <=
```

Having commands that act on objects will require us to have a means of manipulating those objects: we need an analogue to the regular expressions and helper tools for text manipulation. When dealing with objects, we have to provide instructions using a programming language that will allow us to access, create, and modify objects. PowerShell lets us wrap those instructions in script blocks. Blocks are objects, similar to procedures or functions. A block's body contains one or more instructions, which can be executed several times with different parameters.

Processing Collections Using `ForEach-Object`

PowerShell provides language features for iterating over collections using the `for` operator. However, its imperative nature often makes iteration clumsy and error prone, as we usually have to repeat the same loop code over and over again. The `ForEach-Object` cmdlet minimizes the repetition by applying an action to each object, collecting the results into a new collection.

At a bare minimum, `ForEach-Object` requires two inputs: a collection, usually piped in from the previous command, and an action. Actions are expressed as script blocks. To paraphrase the old proverb, an example is worth a thousand words, so here is how we can get the file sizes for all files in the current folder by passing a script block that returns the `Length` property for a given object:

```
PS> dir
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\PowerShell\tmp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	7/7/2007 11:33 PM	4	test.txt
-a---	7/7/2007 11:34 PM	12	test2.txt
-a---	7/7/2007 11:33 PM	4	test3.txt
-a---	7/7/2007 11:34 PM	14	test4.txt

```
PS> dir | ForEach-Object { $_.Length }
```

```
4
12
4
14
```

Note the `$_` special variable! Other than bringing tears to Perl fans' eyes, it serves an important purpose: by convention, it keeps a reference to the current object. `ForEach-Object` works by looping over all objects, setting up the `$_` variable and calling the script block.

Quite often, an iteration process needs some initialization and a good finish after we are done with all objects. We can accomplish that by passing two script blocks as the `begin` and `end` parameters. Here is an ad hoc reporting technique that goes over all files in the current directory:

it begins by initializing an accumulator variable to zero, iterates adding up file sizes inside, and ends by displaying the sum to the user:

```
PS> dir | ForEach-Object -begin { $sum = 0 } -process `
    { $sum += $_.Length } -end { echo "Total: $sum bytes." }
```

Total: 34 bytes.

The process parameter is the default one. Though the process parameter is not usually necessary, many people find that, when providing begin and end blocks, including it makes the command more readable.

Filtering Collections Using Where-Object

Very often, the built-in cmdlets provide more data than we need. Take `Get-Process` for example: it will return all processes and allow for rudimentary filtering by process name only. What if we want to get all processes that occupy more than 20MB of RAM? We have to pipe the output of `Get-Process` into `Where-Object` (I will use the shorthand `where` in this book. The other alias, `?`, feels too cryptic and might inhibit readability). Can you guess the essential parameters that `where` needs? That's right—it takes a collection and a script block at minimum. The script block is executed against all objects and only the ones that evaluate to `$true` will be included in the resulting collection.

```
PS> Get-Process | Where-Object { $_.WS -ge 20MB }
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
156	8	99296	58544	191	134.00	2808	dwm
847	39	59112	58472	277	99.48	2852	explorer
431	11	65792	53012	157		1004	svchost
328	17	21780	48908	297	23.19	2600	WINWORD

The `WS` property is an alias for `WorkingSet`, which is the amount of physical memory the process occupies at the moment. The `-ge` switch is the greater-than-or-equal comparison operator. Most languages use `<`, `>`, `<=`, `>=` to compare values. In PowerShell, however, the `>` and `<` symbols are reserved for input and output redirection, so the language designers had to pick something else. For in-depth coverage of logical operators and comparisons, please refer to Chapter 2.

Adding or Deleting Properties Using Select-Object

`Select-Object` has also been named after a related operation from the databases world—the `SELECT` clause that defines the columns that are returned in our query result set. While `Where-Object` allows us to select a subset of a collection's items, analogous to selecting rows from a table, `Select-Object` does the same for properties, similar to manipulating a table's columns. The `Select-Object` cmdlet allows us to remove properties from the resulting objects or even add new ones that will be calculated on the fly.

At the very minimum, `Select-Object` asks for a list of properties that will be returned. Here is how to get just the names and last access times for the files in the current folder:

```
PS> dir | Select-Object Name,LastAccessTime
```

Name	LastAccessTime
----	-----
test.txt	9/7/2007 8:55:12 AM
test2.txt	9/7/2007 8:55:12 AM
test3.txt	9/7/2007 8:55:12 AM
test4.txt	9/7/2007 8:55:12 AM

Adding properties to objects is done in the same way. We have to follow the convention of defining a calculated property and pass a dictionary that contains the property's name and the expression that calculates the property. Here is how we can add the `LastAccessWeekDay` property to our file list:

```
PS> dir | Select-Object Name,@{Name="LastAccessWeekDay";
    Expression={$_.LastAccessTime.DayOfWeek}}
```

Name	LastAccessWeekDay
----	-----
test.txt	Friday
test2.txt	Friday
test3.txt	Friday
test4.txt	Friday

Note that `Select-Object` follows the convention of passing the current object as the `$_` variable.

Another thing worth mentioning here is that `Select-Object` has a rudimentary mechanism of selecting the first *N* rows (or the last *N* for that matter) by passing a number as the `first` and `last` parameters. Here is how to use that to get the first or the last two text files in a folder:

```
PS> dir *.txt | select -first 2
```

Directory: Microsoft.PowerShell.Core\FileSystem::C:\PowerShell

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	7/7/2007 11:33 PM	4	other.txt
-a---	7/7/2007 11:33 PM	4	test.txt

```
PS> dir *.txt | select -last 2
```


381	11	34492	1132	1520	1904 sqlservr
713	0	0	532	4	4 System
28	1	248	524	4	364 smss
0	0	0	16	0	0 Idle

Very often, you'll want to strip out duplicate objects from your data. You can use `Get-Unique` to do that. Like most operations that strip duplicates, `Get-Unique` requires that the input collection be sorted. The algorithm it uses internally is to traverse the collection and remove any item that is the same as its predecessor—that will not work if the collection is not sorted and equal items do not appear next to each other. Here is how to get the owners for all files in the current folder:

```
PS> dir | Get-Acl | select Owner | sort Owner | `
    Get-Unique -asString
```

```
Owner
-----
NT AUTHORITY\SYSTEM
NULL\Hristo
```

We get the access control list (ACL) for every file, select just the owners, sort by those value, and then get only the unique entries. We want to get objects with unique string representations, so we pass the `asString` parameter.

It is a little known fact that different cultures have different comparison rules. Among the classic examples of this fact are diacritic characters in strings that are compared according to special rules in French. That is why `Sort-Object` allows us to pass a `Culture` parameter to be used for sorting. Note the difference in the sort order when using `en-US` to designate American English and when using `fr-FR` to designate French as spoken in France:

```
PS> "cote", "côte", "coté", "côté" | Sort-Object -Culture "en-US"
cote
coté
côte
côté
PS> "cote", "côte", "coté", "côté" | Sort-Object -Culture "fr-FR"
cote
côte
coté
côté
```

The `Culture` name is the .NET culture name, where the first part is an ISO 639 language name and the second, an ISO 3166 country or region code.

Pipeline Tees

Building long command pipelines is a very powerful way to express a complex action. Sometimes, though, we have to get the objects produced by a command somewhere in the middle of our pipeline and store them for future reference. The most common scenario is logging: we need to store a list of objects before acting on them. Say we want to terminate a couple of processes,

but we need to write out the victim processes' names to an improvised audit log. Normally, we would get and terminate the processes using `Get-Process` and `Stop-Process` like this:

```
PS> Get-Process notepad | Stop-Process
PS>
```

To write the objects that `Get-Process` returned to a file, we need to inject a `Tee-Object` call in the pipeline:

```
PS> Get-Process notepad | Tee-Object -file kill.log | Stop-Process
PS> type kill.log
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
48	3	1232	7568	59	0.16	3488	notepad

Sometimes, we do not need to write to disk at all, and a variable is just fine for storing intermediate results. We can do that by using the variable parameter:

```
PS> Get-Process notepad | Tee-Object -variable victims | Stop-Process
PS> $victims | select ProcessName,HasExited
```

ProcessName	HasExited
notepad	True
notepad	True

Note that we should not use the dollar sign (\$) variable prefix when passing the variable name to `Tee-Object`. We pass the variable name, and the cmdlet defines a variable and sets its content.

Grouping Objects

Dealing with many objects often involves classifying them into several groups. PowerShell makes that very easy to do with its `Group-Object` cmdlet. The simplest and most common scenario is to split a collection into groups according to a property value. Here is how we can get a report of all services split into groups according to their status:

```
PS> Get-Service | Group-Object Status
```

Count	Name	Group
73	Running	{AeLookupSvc, Appinfo, AudioEndpointBuil...
79	Stopped	{ALG, AppMgmt, aspnet_state, Browser...}

The result is a collection of `GroupInfo` objects. We can get each of them and query it for the actual objects that belong to the group by accessing the `GroupInfo` `Group` property:

```
PS> (Get-Service | Group-Object Status)[1].Group
```

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

The preceding example will show us all services in the “Stopped” group, which contains all stopped services. Note that use of parentheses to terminate the pipeline result and the use of that result as a regular variable that contains a collection.

Gathering Collection Statistics

Commonly, when dealing with a big data set, we need to gather statistics that involve querying every object. This is where the `Measure-Object` cmdlet comes into play. It will go over a collection and calculate several of the most common statistics for a property value: count, sum, minimum, maximum, and average—with a minimal amount of code. As an example, let’s collect statistics about all file sizes:

```
PS> dir -r | Measure-Object -property Length -min -max -average -sum
```

```
Count      : 2565
Average    : 55113.0101364522
Sum        : 141364871
Maximum    : 19622912
Minimum    : 0
Property   : Length
```

The preceding query finds all files recursively by using the `dir` command’s `-r` switch. It then loops over the result, queries each file’s `Length` property, and calculates the statistics.

An interesting thing about `Measure-Object` is that it can be completely replaced with a properly crafted `ForEach-Object` command that sets up a global variable and updates it with all objects’ values. Of course, that would require a lot more typing, so you’re better off using `Measure-Object` when you can, as a convenience, and leaving `ForEach-Object` for more advanced scenarios.

Detecting Changes and Differences Among Objects

All scripting, sooner or later, will hit the task of comparing two objects or collections, trying to find what has been added or deleted. `Compare-Object` is our friend here: it knows how to loop over two objects, the reference object and another one, and report on their differences. Traditional `diff` tools work on text only, while `Compare-Object` works on real objects. Here is how we can use it to detect the changes between two folders: the current one and yesterday’s backup:

```
PS> diff (dir PowerShellBackup) (dir PowerShell)
```

```
InputObject                               SideIndicator
-----
kill.log                                  =>
```

The => side indicator means that the kill.log object is present in the right-hand collection, the difference object only. An indicator of <= means the same about the left-hand collection, the reference object.

Compare-Object can help us build some really powerful scripts. For example, we can pipe its output to ForEach-Object and build a simple backup updating script that will copy newly created files from PowerShell to PowerShellBackup. Copying all files from one folder to another is a risky operation, and just to make sure we do not overwrite an important file and lose its contents, we pass the -confirm switch to the copy command. Here is the code:

```
PS> diff (dir PowerShellBackup) (dir PowerShell) | `
    where { $_.SideIndicator -eq ">" } | `
    foreach { copy -confirm "PowerShell\$(($_.InputObject)" `
        "PowerShellBackup" }
```

Confirm

Are you sure you want to perform this action?

Performing operation "Copy File" on Target "Item:

C:\PowerShell\PowerShell\kill.log Destination:

C:\PowerShell\PowerShellBackup\kill.log".

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help

(default is "Y"):y

Note that we are filtering the diff results first and get only objects that have a SideIndicator of =>. We then use the \$_.InputObject value to build a path that we can copy.

The Object Pipeline and Functional Programming

The object pipeline is the single PowerShell feature that changes the entire paradigm of shell scripting, because it allows you to work directly with objects, making interacting with various commands a lot easier than other shells do.

Passing objects around is not the only important thing though. The code name for the early, unofficial PowerShell releases was Monad. The name comes from a philosophy called Monadism. The core idea of Monadism is that every material in the world is composed of simple nondivisible parts; everything complex, valuable, and beautiful is composed of simple parts. This is the guiding principle of the PowerShell architecture: we get complex behavior by combining simple commands. Large scripts and commands emerge by chaining commands and extending each command's behavior by providing other commands in the form of script blocks.

The object pipeline, then, is not just about objects in the pipeline. You also should take special notice of the script block objects that are passed to the cmdlets you have seen so far. The pipeline grows even more powerful when it has the ability to apply script blocks to objects that it processes. The style of programming that passes executable code objects to other code

has much in common with the functional programming paradigm where functions are first class objects and can be created and passed around at runtime. PowerShell script blocks are like functions that we create on the fly and pass to commands in order to extend their behavior. This is where the power of cmdlets such as `ForEach-Object` and `Where-Object` comes from. At their core, those two commands are implemented as a loop that iterates over a collection and creates another collection. Their value comes from being able to apply the user defined action to objects in that collection. If we did not have script blocks to abstract the action, we would be forced to write every collection iteration loop by hand. To get specifics and advanced script object usage examples, see Chapter 4.

Summary

This chapter has given you the basic tools for working with collections by setting up object pipelines, so that you can transform data into something that works for you. It is important to start with those operations so that we can use them later, on objects that come from various sources. Every PowerShell command is targeted at processing a collection of objects and will return another collection and any single object will transparently be converted to a collection when needed.

Most useful PowerShell scripts start out as experiments performed on the command line: Can I get the list of processes running on that remote machine? Can I filter out the ones I do not need? Can I sort the result by memory usage? Can I get the top ten memory hogs? Can I kill the worst offender? Most of these queries can be performed in a single pipeline: probably all of them could be done in a single pipeline, but sometimes it is best if we split up operations for readability. You will see details on taking the next step and saving pipelines and other commands to script files in Chapter 8.

As a last piece of advice, I would recommend learning the aliases for the commands discussed in this chapter. They will help when trying things out quickly on the command line. We will learn more aliases and how the command alias system works in general in Chapter 6.



Working with Script Blocks

The PowerShell programming language is very dynamic. Until now, I have focused on its abilities to manipulate objects of any kind. In this chapter, we will explore my favorite language feature: script blocks. Blocks help craft elegant code not only when manipulating collections of objects but in many other situations. Most importantly, script blocks allow us to encapsulate pieces of code and delay their execution. Script blocks are containers holding pieces of code that do not have to be assigned a formal name. We can create them dynamically in the middle of any script operation, and, of course, we can execute them multiple times, passing different parameters.

You have already seen how you can use them to specify an action that a cmdlet will execute on our behalf when traversing a collection of objects. Now, we will focus on using them to encapsulate an action that can be executed at a later time. This programming technique offers great power and expressiveness, and we will use it to make our code simpler, more concise, and, of course, more readable.

Defining Script Blocks

Defining a script block literal is as simple as surrounding several program statements with curly braces. The statements will not be executed immediately; instead, a new script block object will be created and returned. Here is our first script block:

```
PS> { Write-Host "Hello from a script block" }  
Write-Host "Hello from a script block"
```

As you can see, the actual `Write-Host` statement did not execute. We got back a script block object whose string representation is the script code contained inside the block. There is little value in creating blocks if we do not keep references to them so that we can use them later. Here is how to assign a newly created block to a variable:

```
PS> $helloBlock = { Write-Host "Hello from a script block" }  
PS>
```

Let's look under the hood and see what a script block variable really contains:

```
PS> $helloBlock.GetType().FullName  
System.Management.Automation.ScriptBlock
```

As you can see, script block objects are instances of `System.Management.Automation.ScriptBlock`, which is a normal .NET class that is a part of the PowerShell infrastructure. Therefore, we can treat script blocks as .NET objects, and here is how we can get their properties and methods:

```
PS> $helloBlock | Get-Member
```

```
TypeName: System.Management.Automation.ScriptBlock
```

Name	MemberType	Definition
Equals	Method	System.Boolean Equals(Object obj)
GetHashCode	Method	System.Int32 GetHashCode()
GetType	Method	System.Type GetType()
get_IsFilter	Method	System.Boolean get_IsFilter()
Invoke	Method	System.Collections.ObjectModel.Collection`1[...]
InvokeReturnAsIs	Method	System.Object InvokeReturnAsIs(Params Object...)
set_IsFilter	Method	System.Void set_IsFilter(Boolean value)
ToString	Method	System.String ToString()
IsFilter	Property	System.Boolean IsFilter {get;set;}

We got back a pretty interesting list of properties that can give us a clue of how PowerShell uses blocks: it creates a `ScriptBlock` object at the time it parses the script code and calls its `Invoke` or `InvokeReturnAsIs` methods at the time of execution.

Now that you know how to create script blocks, we can start using them. We can invoke a script block by prefixing it with the invoke operator (&). Here is how to do that with a variable that contains a block:

```
PS> $helloBlock = { Write-Host "Hello from a script block" }
PS> &$helloBlock
Hello from a script block
```

The invoke operator works with more than variable definitions; it is an expression operator, and you can use it on script blocks that are declared inline. For example, the following statement defines an anonymous script block and executes it immediately:

```
PS> & { Write-Host "Hello from a script block" }
Hello from a script block
```

Blocks are compiled objects that can be passed around and executed multiple times. They are reference objects, and assigning a variable to a block just makes that variable point to the block reference in memory. To illustrate that, let's execute a block several times, accessing it from different variables:

```
PS> $block1 = { Write-Host "Block executed" }
PS> $block2 = $block1
PS> for ($i = 0; $i -lt 3; $i++)
```

```
{
    &$block1
    &$block2
}
```

```
Block executed
Block executed
Block executed
Block executed
Block executed
Block executed
```

Look at the six `Block executed` messages! Our loop body executes three times, each time executing the blocks that `$block1` and `$block2` point to. Both variables point to the same script block object, so our block was executed six times.

Passing Parameters and Returning Values

None of the blocks that you have seen so far executed an action that interacted with the outside world in any sophisticated way. Now that I've mentioned expressions, we can look at blocks from the perspective of an operation that can be used in an expression. Operations take parameters and return values after they are evaluated. Let's start with return values.

Returning a value from a script block simply requires that you output an object that will not be consumed by a cmdlet or another expression. Here is a block that returns a number:

```
PS> $number = { 5 }
PS> &$number
5
PS> 1 + (&$number)
6
```

As you can see from the last command in the previous example, the returned number can be used in expressions under one condition: you have to use parentheses. If you do not, the plus sign (+) operator mistakenly assumes the ampersand (&) is its right-hand operand and throws an error:

```
PS> 1 + &$number
You must provide a value expression on the right-hand side of the '+' operator.
At line:1 char:4
+ 1 + <<<< &$number
```

Note that outputting an object does not end the execution inside the block, and the rest of the statements after the output object still get executed. For example, you can print a string to the console window after returning a value:

```
PS> $numberPrint = { 5; Write-Host "generated a number" }
PS> &$numberPrint
5
generated a number
PS> $result = &$numberPrint
generated a number
PS> $result
5
```

The `$numberPrint` block both returns a value and writes a string to the console. Looking at the anonymous block invocation, you might conclude that the block returns or prints two variables, but that is not the case. Look at the assignment operation that follows: the `$result` variable gets assigned the number 5, the real return value, while the string is printed to the console.

To end the execution and exit the script block, you can use a `return` statement. It terminates execution and returns the value. Changing the preceding block to use `return` will prevent the `Write-Host` command from executing:

```
PS> $number = { return 5; Write-Host "generated a number" }
PS> $a = &$number
PS> $a
5
```

You can use the `return` statement to simply exit a block and stop execution. It does not require you to provide a value. If you omit the value, the script block will simply exit without returning a value. In that case the block will return a value to its caller only if something has been output before the `return` statement. Here are two examples that illustrate that:

```
PS> $noReturn = { return; 4 }
PS> &$noReturn
PS> $numberReturn = { 4; return }
PS> &$numberReturn
4
```

The first script block never gets to return the number 4, because execution stops with the `return` statement. The second block returns the number 4, because it has been output before the `return` statement.

You might be wondering why the second script block even worked. To tell the truth, this is an area where PowerShell differs from many programming languages: its script blocks can return multiple values. That is why we referred to returning values as outputting values—you can do that multiple times, and the result will be an array. As an illustration, let's take a look at a block that returns three numbers:

```
PS> $numbers = { 1; 2; 3; }
PS> &$numbers
1
2
3
PS> (&$numbers).GetType().FullName
System.Object[]
```

Note that we do not explicitly return an array—if we wanted to do that, we would have to use commas as number separators instead of semicolons. The semicolon is a statement terminator, and as you can see, the result of returning three numbers in a row is an array.

A useful script block would need a way to get parameters from the outside world so that it can be executed with different data. Parameters can be passed and interpreted according to their positions. A block will always have a predefined `$args` variable automatically set up, which will hold a collection of the supplied parameters. We can use it to create a block that outputs a customized message to the user given a first and last name:

```
PS> $greeter = {  
    $firstName = $args[0]  
    $lastName = $args[1]  
    Write-Host "Hello, $firstName $lastName"  
}
```

```
PS> &$greeter "John" "Smith"  
Hello, John Smith  
PS> &$greeter "Al" "Bundy"  
Hello, Al Bundy
```

Accessing parameters by index is a fine technique and is the only one available in many programming languages. Unfortunately, doing so becomes too error prone once the parameters grow in number. It is best to use that in either very simple scenarios that cannot possibly go wrong or in advanced scenarios when you do not know the number of parameters at the time we define our script blocks. Simple and advanced scenarios aside, in most cases we can use named parameters by declaring them inside script blocks using the `param` keyword. Here is how the previous example looks after switching to named parameters:

```
PS> $greeter = {  
    param ($firstName, $lastName)  
    Write-Host "Hello, $firstName $lastName"  
}
```

```
PS> &$greeter "John" "Smith"  
Hello, John Smith
```

This feels much better and more elegant. To make invocations almost bulletproof, we can use a switch-based syntax to associate parameter names with parameter values at the time of invocation. This way, we do not have to remember the parameter order:

```
PS> &$greeter "John" "Smith"  
Hello, John Smith  
PS> &$greeter -firstName "John" -lastName "Smith"  
Hello, John Smith  
PS> &$greeter -lastName "Smith" -firstName "John"  
Hello, John Smith
```

As you can see, you can provide parameters in any order you wish, even if you can't remember the full parameter names. PowerShell will find the parameters for you like so:

```
PS> &$greeter -last "Smith" -first "John"
Hello, John Smith
PS> &$greeter -l "Smith" -f "John"
Hello, John Smith
```

Note that usually the first letter of a parameter name will suffice. In cases where multiple parameter names start with the same character sequence, you will have to provide the first several characters that uniquely identify the parameter.

You can use the `param` statement to convert the parameters to a type you need, so that if the caller provides a value that can be converted to the specific type, it will get converted for us. Another good thing is that we will get an explicit error if the value cannot be converted successfully. That feature alone greatly simplifies error handling. Let's see how to use parameter types to define a block that sums two numbers:

```
PS> $sum = {
    param ([int] $a, [int] $b)
    $a + $b
}

PS> &$sum 3 4
7
PS> &$sum "3" 4
7
PS> &$sum "not a number" 4
param([int] $a, [int] $b) $a + $b
: Cannot convert value "not a number" to type "System.Int32". Error: "Input string was not in a correct format."
At line:1 char:2
+ &$ <<<< sum "not a number" 4
```

The `$sum` block accepts two integers. Passing a string will trigger a conversion. If it is successful, everything continues as planned. If unsuccessful, we get a descriptive error that politely hints we are trying to fit a square peg into a round hole.

What happens when the script block caller does not provide all parameters? By default, missing parameters get initialized to `$null`, and we can use that fact to check if a parameter has actually been provided. Going back to the original greeting example, let us provide an "Unknown" value for the `$lastName` parameter if it has not been provided by the script caller:

```
PS> $greeter = {
    param ([string] $firstName, [string] $lastName)
    if (!$lastName)
    {
        $lastName = "Unknown"
    }
    Write-Host "Hello, $firstName $lastName"
}
```

```
PS> &$greeter "John" "Smith"
Hello, John Smith
PS> &$greeter "John"
Hello, John Unknown
```

Providing default values for parameters like that can quickly bore you to death, as the `if` statement shown previously would have to be repeated for every parameter throughout your code. Fortunately, PowerShell provides a shorthand notation—parameter default values. Assigning a value in the `param` block means that the parameter will be initialized with this value if the caller has not provided a value. Let's rewrite the `$greeter` block using a default value for `$lastName`:

```
PS> $greeter = {
    param ([string] $firstName, [string] $lastName = "Unknown")
    Write-Host "Hello, $firstName $lastName"
}

PS> &$greeter "John" "Smith"
Hello, John Smith
PS> &$greeter "John"
Hello, John Unknown
```

I like the result; it is much more concise and readable.

Parameter default values can contain any expression. We can exploit this fact to implement mandatory parameters. Mandatory parameters are parameters that absolutely have to be provided when calling the script block. To do that, we add an expression that throws an exception when evaluated. In that way, the exception will be thrown if the caller fails to provide the parameter. Let's use this technique in our `$greeter` block and make our `$firstName` parameter mandatory:

```
PS> $greeter = {
>> param ($firstName = $(throw "firstName required!"), $lastName)
>> Write-Host "Hello, $firstName $lastName"
>> }
>>

PS> &$greeter -lastName "Smith"
firstName required!
At line:2 char:28
+ param ($firstName = $(throw <<<< "firstName required!"), $lastName)
```

For now, you can just follow the `$(throw "Error message")` pattern to implement mandatory parameters. We will discuss exceptions and error handling in detail in Chapter 9.

Processing Pipeline Input

Until now, we have used blocks to encapsulate executable actions and expressions. In the previous chapter, you saw how you can pass script blocks to cmdlets and invoke them for every item in the pipeline. To refresh your memory, look at the following example, which passes a block to the `ForEach-Object` cmdlet to get the last write dates for all text files:

```
PS> dir *.txt | ForEach-Object { $_.LastWriteTime.Date }
```

```
Friday, September 14, 2007 12:00:00 AM
```

```
Thursday, September 13, 2007 12:00:00 AM
```

```
Saturday, July 07, 2007 12:00:00 AM
```

Now, let's dig deeper! Cmdlets are not the only objects that can work with the input pipeline—we can make script blocks do that too. Each script block can contain three special sections that help us work with pipeline input: `begin`, `process`, and `end`. Each section looks like a nested block and contains zero or more statements. The most important section is the `process` one, as it is invoked for every object in the pipeline. By convention, the current object is passed via the `$_` special variable. Using a `process` section, we can easily write a script block that will filter a collection of numbers and return the ones greater than five, for example:

```
PS> $greaterThanFive = {  
    process {  
        if ($_ -gt 5)  
        {  
            return $_  
        }  
    }  
}
```

```
PS> 3,4,5,6,7 | &$greaterThanFive
```

```
6
```

```
7
```

Similar to `process`, we can embed `begin` and `end` sections. These two are executed before pipeline processing starts and after it finishes, respectively. They are useful in cases where we want to calculate collection statistics or maintain some state that will help us build the return result. As an example, let's see how to create a script block that sums all numbers that have been piped in:

```
PS> $sum = {  
    begin {  
        $total = 0  
    }  
    process {  
        $total += $_  
    }  
    end {  
        return $total  
    }  
}
```

```
PS> 1,2,3,4 | &$sum
```

```
10
```


This time, our process section only adds the current value to the running total, and the act of returning the value is performed in the end section. The actual addition operation is worth noting here: we are using the `+=` operator, which takes the right-hand variable value and adds it to the left-hand variable. We calculate the total by repeatedly adding the current value, passed in the `$_` variable, to the `$total` variable.

Script blocks can accept both parameters and pipeline input. Named and position parameters are available in the current scope and in the `$args` collection, while pipeline objects are passed in `$_`. To demonstrate how to set up both parameters and process pipeline input, we can write a script block that will add a number to all items in the pipeline:

```
PS> $adder = {  
    param ($number)  
    process {  
        $_ + $number  
    }  
}  
  
PS> 1,2,3 | &$adder 10  
11  
12  
13
```

Numbers are fine to make a point, but let's try a real-world example. We can create a block that will calculate the number of days between a specific date and the last write date for all text files in the current folder:

```
PS> $dateDiff = {  
    param ([datetime] $referenceDate)  
    process {  
        ($referenceDate - $_.LastWriteTime.Date).Days  
    }  
}  
  
PS> dir *.txt | &$dateDiff ([datetime]::Today)  
5  
6  
74
```

Note that I've surrounded `[datetime]::Today` with parentheses. If we don't do that, the `DateTime` object will be converted to a string and will be parsed back into a `DateTime`. This is both inefficient and likely to raise an exception as the string representation of a date is not guaranteed to be parsed back correctly.

Variable Scope

The scope of a variable is the portion of a program's code that can use a variable. PowerShell has quite sophisticated scoping rules for named objects, like variables, that kick in when we start to send them across script block, function, and script boundaries. I will start the discussion

here with an emphasis on script blocks, and I will cover functions and script files in Chapters 5 and 8 respectively.

DEFINING VARIABLE SCOPE

Scope is a computer science term that defines the surface area in which a variable is visible. It can be likened to the idea of a namespace—the mechanism that allows us having objects with the same name existing in different environments. Very simple languages do not have the concept of different scopes for different variables; in a sense, every variable is visible everywhere. This quickly leads to problems, because you have to be extra careful when defining variables, so that they do not overwrite other variables. Imagine the following situation:

```
$myVariable = 3
#. . . a lot of code in between . . .
$myVariable = 5
```

When the second variable is assigned, we would have lost the first variable, and that means we would introduce a hard-to-catch bug.

Modern programming languages, PowerShell included, solve the problem by allowing every logical code unit to have a different space for variables. This way, initializing and assigning a variable in one code unit will not affect other units. Script blocks are such units and so are functions and script files, so assigning variables from within a script block will not overwrite variables with the same names that have been declared outside that script block.

```
$myVariable = 4
& {
    $myVariable = 5
}
```

In the preceding example, the script block gets a different `$myVariable` variable, so that it does not overwrite the one declared outside the block. Inside the script block, `$myVariable` will refer to the variable containing the number 5. Outside the block, the `$myVariable` variable will still contain the number 4.

Scopes are hierarchical in nature. Each script block starts a new local scope and can declare its own variables. In addition, it inherits the variables declared in the parent scope. The parent scope is usually the global one, but this is not always the case. Nesting a script block inside another block, for example, will have the parent scope refer to the outer script block. A block can freely access variables declared in parent scopes, but most of the time, that access is read only. That is, a block can see and read a variable declared before it, but extra effort is required if it has to modify it. The theory is easier to grasp with an example, so here is a script block that reads a variable from the parent scope and writes it to the console:

```
PS> $name = "John"
PS> $personAction = { Write-Host "Hello $name" }
PS> &$personAction
Hello John
```

Changing the `$name` variable and calling the block again will logically result in a different message:

```
PS> $name = "Mike"
PS> &$personAction
Hello Mike
```

What happens if we try to change the variable from within the block? To assign a value to a variable, PowerShell will first look to see if it is declared in the current scope and, if it is not, will create a new local variable. Here is what happens if we try to assign the `$name` variable from within the `$personAction` block:

```
PS> $personAction = {
    $name = "Scott"
    Write-Host "`$name changed to $name"
}

PS> $name = "John"
PS> &$personAction
$name changed to Scott
PS> $name
John
```

There are two important things to note in the preceding snippet. First, the block uses the modified variable and gets “Scott” as the person’s name. Second, the `$name` variable inside the block is not the same as the `$name` variable declared outside. In fact, they are two completely different variables, and the local variable shadows the global one. That is why the global `$name` variable remains untouched and preserves its original value after the block is executed. This suggests that we cannot change a global variable from within a script block. Actually, we can, but we have to work harder.

We can change variables in all scopes once we realize that the dollar sign syntax for variables is just a convenient shorthand for calling the `Get-Variable` and `Set-Variable` cmdlets behind the scenes. Using `Get-Variable`, we can obtain a reference to a `PSVariable` object that has the name and the value for a variable. Here is how we can use that to get our `$name` variable:

```
PS> Get-Variable name
```

Name	Value
----	-----
name	John

Most of the time, we know the variable name (after all we pass it as a parameter to `Get-Variable`), and we are interested in the value only. We can pass the `-valueOnly` parameter and get straight to the object that the variable contains:

```
PS> Get-Variable name -valueOnly
John
```

The Set-Variable cmdlet, on the other hand, finds variables using their names and modifies their values. We can use it as a replacement for the assignment operator and assign “Mike” to the \$name variable:

```
PS> Set-Variable name "Mike"
PS> $name
Mike
```

Now, let’s rewrite the \$personAction example using Get-Variable and Set-Variable:

```
PS> Set-Variable "name" "John"
PS> Get-Variable "name" -valueOnly
John
PS> $personAction = {
    Set-Variable name "Scott"
    Write-Host "`$name changed to $name"
}

PS> &$personAction
$name changed to Scott
PS> Get-Variable "name" -valueOnly
John
```

The Get-Variable and Set-Variable syntax feels a lot clumsier than the assignment syntax we are used to, but it is more powerful. Both Get-Variable and Set-Variable support a -scope parameter that tells them where to look for the variable. A scope of 0 means the current scope. A value of 1 means the parent scope, 2 the grandparent one, and so on. We can use that to modify a variable in the parent scope from within a script block. Here is how we can change \$personAction to do that:

```
PS> $name = "John"
PS> $personAction = {
    Set-Variable name "Scott" -scope 1
    Write-Host "`$name changed to $name"
}

PS> &$personAction
$name changed to Scott
PS> $name
Scott
```

As you can see from the last statement, this time, the \$name variable in the parent scope changed to “Scott”.

Most of the time we do not deal with deeply nested scopes and are interested in distinguishing variables in the current scope and the global one. We can take advantage of a special syntax that uses a prefix before variable names to denote their scope. In the \$personAction example, we can declare the \$name variable to be global, which would enable writing to it from within the script block. Here is how to do that:

```

PS> $global:name = "John"
PS> $personAction = {
    $global:name = "Scott"
    Write-Host "`$global:name changed to $global:name"
}

PS> &$personAction
$global:name changed to Scott
PS> $global:name
Scott

```

We prefix the variable name with `$global:` to make it a global variable. Note that we do use one dollar sign before “global” and do not add another before the actual variable name.

PowerShell supports four scope prefixes:

- `global`: This scope has the greatest visibility. All code sees global variables.
- `script`: This is similar to global but is limited to the current script file. A variable like `$script:name` will be visible throughout the current script file but not outside it. I will explain script files in Chapter 8.
- `local`: This is the default scope. Variables will be visible in the current and nested scope. The assignment syntax can modify a variable in only the current local scope.
- `private`: This is the most restricted scope. Variables will be visible in only the current scope. This is convenient if you want to hide a variable from a child.

The private scope is very interesting. Let’s see how to hide a variable from a script block:

```

PS> $private:numberValue = 1
PS> $checkNumberValue = {
    if ($numberValue -eq $null)
    {
        Write-Host "Cannot access `$numberValue"
    }
}

PS> &$checkNumberValue
Cannot access $numberValue
PS> $numberValue
1

```

The variable remains visible to the parent scope. Of course, even declaring the variable private will not hide the variable from the almighty `Get-Variable` and `Set-Variable` cmdlets:

```

PS> $checkNumberValue = {
    if ((Get-Variable "numberValue" -valueOnly -scope 1) -ne $null)
    {
        Write-Host "Can access `$numberValue from parent scope"
    }
}

```

```
PS> &$checkNumberValue
Can access $number from parent scope
```

When resolving a reference to a variable without a scope prefix, PowerShell starts looking in the private and local scopes and then continues in the script and global ones. Note that this holds true for reading a variable's value. Assigning a new value without specifying a prefix will always create a local variable. Here is an example:

```
PS> $global:name = "John"
PS> $personAction = {
    Write-Host "`$name = $name"
    $name = "Scott"
    Write-Host "`$name changed to $name"
}

PS> &$personAction
$name = John
$name changed to Scott
PS> $name
John
```

The inner block reads `$name` and gets the global variable. It then assigns a value, thus creating a local variable. At the end, the global variable remains unchanged.

PowerShell supports a special syntax for executing a command in the current scope by prefixing it with a single dot. This is referred to as dot-sourcing a command. Dot-sourcing a script block makes its inner variables available to the current scope. To illustrate that, let's look at a block that declares an inner variable:

```
PS> $numberBlock = {
    $numberInBlock = 1
    Write-Host $numberInBlock
}

PS> &$numberBlock
1
PS> $numberInBlock -eq $null
True
```

As you can see, the `$numberInBlock` variable is available inside the block only. Let's see what happens when we dot-source the block:

```
PS> . $numberBlock
1
PS> $numberInBlock
1
```

The `$numberInBlock` variable is now available, because `$numberBlock` has been executed in the current scope. Note that we do not need an ampersand (&) symbol before the `$numberBlock` variable reference to execute it if we use dot-sourcing.

Scoping gets even more interesting when pipelines are involved. Blocks in the same pipeline share a single scope. Here is a pipeline that contains two blocks passed to cmdlets applied at a different point in the pipeline. The second block sees variables declared in the first one:

```
PS> dir *.txt | foreach { $file = $_.Name; return 3 } |  
    foreach { "got $file" }  
  
got test.txt  
got test2.txt  
got test3.txt  
got test4.txt
```

Note that the first `foreach` block only sets the `$file` variable and returns a dummy number. The `$file` variable is available to the second `foreach` block that actually returns it. This is very convenient if you wish to store some information at an early stage in the pipeline and have it available at later.

Invoking Strings as Expressions

Almost all script languages have a special function or operation that takes a string as an input parameter, compiles it, and executes it as a piece of code. Other popular languages such as JavaScript call that function `eval`. PowerShell's equivalent is the `Invoke-Expression` cmdlet. Strings can be either passed as parameters or piped in from another command. Here is how to execute a simple string like "Write-Host 'invoked expression'":

```
PS> Invoke-Expression "Write-Host 'invoked expression'"  
invoked expression
```

Note that we have to escape quotes or use single quotes inside the string that gets executed. This is quite cumbersome, but it is worth the effort. We can manipulate strings at runtime and create highly dynamic and elegant solutions by building commands on the fly. Let's look at how to generate several commands and pipe them to `Invoke-Expression` for execution:

```
PS> 1..3 | foreach { "Write-Host 'Got $_'" } | Invoke-Expression  
Got 1  
Got 2  
Got 3
```

Invoking an expression is quite similar to building and executing script block. In fact, that is precisely what the `Invoke-Expression` cmdlet does under the hood: it creates a new script block, executes it, and pushes any return values down the pipeline for the next command. The only thing to keep in mind is that, unlike a script block, `Invoke-Expression` does not create a child variable scope. It always executes in the current scope. Here is how we can verify that:

```
PS> $name = "John"  
PS> Invoke-Expression "`$name = 'Mike'"  
PS> $name  
Mike
```

As you can see, the `$name` variable got modified, so we must be executing in the current scope. We are using double quotes when constructing the command string, so we have to escape the dollar sign to make the string passed to `Invoke-Expression` contain the `$name` variable reference instead of the expanded variable value.

The best use for `Invoke-Expression` is to easily turn a string into an executable command. It is ideal for quick and dirty solutions that can be crafted in a few minutes. Here is how we can use the cmdlet to build a simple calculator that allows the user to type an expression:

```
PS> $expression = Read-Host "Enter expression"
Enter expression: 3*2+1
PS> $result = Invoke-Expression $expression
PS> Write-Host $result
7
```

Be careful though! Taking user input without verifying its contents and then executing it is a huge security risk and a recipe for disaster. What if the user entered the expression `3*2; del C:\ -recurse -force`? You would have to kiss your files bye-bye. The rule here is to always validate user input and avoid `Invoke-Expression` at all cost when user input is involved. Still, the cmdlet is very handy for your own use when you do not need to strictly check for bad input strings. Remember to be extra careful!

There is another way to execute a string—by using the global `$ExecutionContext` variable. It is of the `System.Management.Automation.EngineIntrinsics` type, and it contains a property, `InvokeCommand`, of the `System.Management.Automation.CommandInvocationIntrinsics` type. We can use `InvokeCommand`'s `InvokeScript` method to do the same thing as `Invoke-Expression` does. Here is how:

```
PS> $ExecutionContext.InvokeCommand.InvokeScript("Write-Host 'invoked'")
Invoked
```

The `$ExecutionContext.InvokeCommand` object is very useful, as it allows us to compile a string into a script block. We can use that to create dynamic commands that we wrap in script blocks for later execution. Here is an example:

```
PS> $cmd = "Write-Host ``$(`$args[0])`"`"
PS> $cmdBlock = $ExecutionContext.InvokeCommand.NewScriptBlock($cmd)
PS> &$cmdBlock "test"
Test
```

Why create script blocks when we can just store the string and use `Invoke-Expression` when we need to execute it? For one thing, the script block solution compiles the script code only once at the time of its creation, while `Invoke-Expression` compiles the script code every time it's executed. In addition, a script block creates a child variable scope for us that is sometimes desirable, as we may want protection against unintentionally modifying a variable in the current scope. As usual, when dealing with nested scopes, declare your parent scope variables as `script` or `global`, or use `Get-Variable` and `Set-Variable` if your script block needs to modify them.

Script Blocks as Delegates

Delegates are an integral part of .NET's event handling mechanism. In short, delegates are objects that keep a reference to a .NET class, a method reference, and an object reference if the method is not static. When a delegate is invoked, it will call the method of the object it references. Events in .NET are implemented as delegates pointing to the event receiver object and method, and passing those delegates to the event sender. The sender invokes the event delegates when it raises the event, and the receiver gets notified.

Script blocks and delegates have a lot in common—both of them point to some code and can be invoked to execute that code. PowerShell's type system allows a script block to be converted to a delegate. There is one caveat though: only delegates of the `System.EventHandler` type are supported. That is, we can convert to delegates only blocks that take an object and a `System.EventArgs` instance as parameters. Let's see how we can declare such blocks and turn them to delegates:

```
PS> $handler = {  
    param ($sender, [EventArgs] $eventArgs)  
    Write-Host "Event raised!"  
}
```

```
PS> $delegate = [EventHandler] $handler  
PS> $delegate.GetType().FullName  
System.EventHandler
```

We can invoke delegates by calling their `Invoke()` method:

```
PS> $delegate.Invoke(3, [EventArgs]::Empty)  
Event raised!
```

There is little value in calling our own delegates to simulate an event, so let's try something real. We can create a small Windows Forms application that can simulate a graphical input box that we can use to ask our users for input. To do that, we need to create a `Form` object, add a `TextBox` and a `Button` control inside and wire the `Button Click` event and make it execute our event handler. Our `Click` event handler will close the form and store the `TextBox` text value in a global variable that we will retrieve later. Here is the code:

```
$null = [Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")  
$form = New-Object Windows.Forms.Form  
$form.Size = New-Object Drawing.Size -arg 300, 85  
  
$textBox = New-Object Windows.Forms.TextBox  
$textBox.Dock = "Fill"  
$form.Controls.Add($textBox)  
  
$button = New-Object Windows.Forms.Button  
$button.Text = "Done"  
$button.Dock = "Bottom"
```

```
$button.add_Click(  
    { $global:resultText = $textBox.Text; $form.Close() })  
$form.Controls.Add($button)  
  
[void] $form.ShowDialog()  
Write-Host $global:resultText
```

See the first line? PowerShell does not load the `System.Windows.Forms` assembly by default, so we need to do it ourselves in order to use the .NET types inside. We create a `Form` object and size it to 300 pixels wide and 85 pixels tall. We next add a `TextBox` control that fills the entire form width; in other words, it stretches horizontally to occupy all available space. After that we add our `Button` control. The really important part is the call to the `add_Click` method. That method gets a delegate as a parameter and attaches it to the `Click` event. Note that we do not even need an explicit cast when calling that method—PowerShell does that for us already. Here is what our input box looks like:

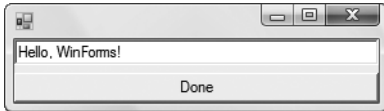


Figure 4-1. *An input box created with Windows Forms*

And this is what we get as a result in our console:

```
PS> Write-Host $global:resultText  
Hello, WinForms!
```

Summary

We have gone through quite a lot in this chapter. Script blocks are an unusual programming construct at first, but fortunately, they are easy to master. Script blocks are only one of PowerShell's mechanisms for abstracting and reusing code, and you will learn the rest of them later in this book.

The next chapter will introduce functions and filters that are closely related to script blocks. Our ultimate goal is to master different code reuse techniques, and we will continue to pursue it in the next chapters by introducing aliases, providers, and script files. Those techniques are important, because they are the stepping stones to building complex scripts.



Working with Functions

The concept of a function in many programming languages, PowerShell included, has much in common with the mathematics concept of a function. A function is an operation that accepts a set of parameters and returns new values. Mathematics requires that a function depend on its parameters only; that is, it should always return the same value given the same parameters. However, shell users do not have to comply with that restriction. A function in PowerShell is just another name for a subroutine or a piece of code that can accept parameters, execute actions, and return values.

Subroutines are possibly the most important invention in computer science and the field of programming. Invented back in 1948 by David Wheeler, Maurice Wilkes, and Stanley Gill as a part of their work on the EDSAC computer, subroutines allow pieces of code to be written separately from the main program and later called when needed. Even today, close to 60 years later, this is how most programs are written. Functions are the primary mechanism for building abstractions and crafting reusable pieces of code in PowerShell. By learning to structure your code with functions, you can make it less complex and more manageable and readable. In this chapter, I will explain both how to create functions and how to use them to solve common problems elegantly.

Defining Functions

A function is very similar to a script block in the sense that it contains executable code. The main difference between the two is that a script block is anonymous—it has to be assigned to a variable before it can be evaluated. Functions get their names at the time of their creation, and that name makes them immediately available for execution. To define a function, we have to use the `function` keyword in the following form:

```
function <name>(<parameter list>)  
{  
    <function body>  
}
```

The function name should start with a letter and can contain any alphanumeric character sequence, as well as hyphens and underscore characters. This is how we can define a simple function that outputs some text:

```
PS> function Say-Hi()
{
    Write-Host "Hello from a function"
}
```

The parameter list parentheses are optional, and we can safely omit them when our function will not accept parameters:

```
PS> function Say-Hi
{
    Write-Host "Hello from a function"
}
```

Defining a function automatically creates an object with its name in the current scope. That makes calling it just a matter of typing its name:

```
PS> Say-Hi
Hello from a function
```

Function Internals

Note that we used a Verb-Noun name for our function, which is the established convention for all commands in PowerShell, for the sake of clarity and readability. Functions behave much like built-in cmdlets, so we name them in the same way.

Functions get registered in the global commands space, and as you already saw, a function's invocation does not differ in any way from a cmdlet's. We can use the `Get-Command` cmdlet to obtain a reference to our function, examine it under our `Get-Member` microscope, and empirically discover some interesting facts about functions. Here is how to get our `Say-Hi` function:

```
PS> $hiFunction = Get-Command Say-Hi -type Function
PS> $hiFunction.GetType().FullName
System.Management.Automation.FunctionInfo
```

We got back a `System.Management.Automation.FunctionInfo` object. Let's see what useful properties it holds for us:

```
PS> $hiFunction | gm -type Property
```

```
TypeName: System.Management.Automation.FunctionInfo
```

Name	MemberType	Definition
----	-----	-----
CommandType	Property	System.Management.Automation.CommandTypes Command...
Definition	Property	System.String Definition {get;}
Name	Property	System.String Name {get;}
Options	Property	System.Management.Automation.ScopedItemOptions Op...
ScriptBlock	Property	System.Management.Automation.ScriptBlock ScriptBl...

The `CommandType` property is already set to `Function` and `Name` is set to “Say-Hi”. The most interesting property is `ScriptBlock`, which tells us what really happens under the hood—functions are simply named script blocks. We can get the script block and even execute it ourselves, without referring to the function. Here is how to do it for our function:

```
PS> $hiFunction.ScriptBlock
Write-Host "Hello from a function"
```

```
PS> & $hiFunction.ScriptBlock
Hello from a function
```

There’s nothing special here—we just use the invoke operator (`&`). You’ve already seen that getting a function’s script block allows us to inspect the code we are about to execute before actually executing it. For convenience, function objects contain the code in the form of a string in their `Definition` property. Here is how to use that and inspect a function’s code to see what it does:

```
PS> $hiFunction.Definition.GetType().FullName
System.String
PS> $hiFunction.Definition
Write-Host "Hello from a function"
```

The `Definition` property holds perfectly valid code that we can manipulate and even execute using the `Invoke-Expression` cmdlet. Here is a sample:

```
PS> Invoke-Expression $hiFunction.Definition
Hello from a function
```

This technique might seem to offer no advantages over getting the function script block and executing it instead, but it might be more convenient in some occasions. You can refer to the previous chapter for the differences between using a script block and a string expression.

Function Parameters

To have a function accept parameters, we can provide a list in the function definition. Here is a sample function that accepts two numbers and writes their sum to the console:

```
PS> function Write-Sum($first, $second)
{
    $sum = $first + $second
    Write-Host "Sum: $sum"
}
```

```
PS> Write-Sum 4 5
Sum: 9
```

Note PowerShell has a different syntax for calling functions than other languages you may be familiar with. Most languages require that you provide parameters inside parentheses and separate one parameter from another with a comma like this: `Write-Sum(4, 5)`. However, that syntax will not work in PowerShell. To quickly get used to the correct PowerShell syntax, start thinking about calling functions as calling external programs or cmdlets and passing positional parameters separated with spaces and using a switch-based syntax for named parameters: `Write-Sum -first 4 -second 5`.

If you miss using parentheses and commas for providing parameters, you can find that notation in the object method call syntax: `$myObject.DoSomething(4, 5)`. Yes, although calling object methods and functions may seem similar, the two have different syntaxes. The only feasible explanation for their differing syntax is that, although they look similar, both operations are quite different in reality.

We can use exactly the same techniques for working with function parameters as we did with script block parameters in the previous chapter—if a parameter-passing technique works for a script block, it will work for a function too. For starters, we can specify a parameter's type, so that the interpreter automatically converts values when passing them to the function. This is how we can force integer parameters for our function:

```
PS> function Write-Sum([int] $first, [int] $second)
{
    $sum = $first + $second
    Write-Host "Sum: $sum"
}
```

```
PS> Write-Sum 2 "2"
Sum: 4
```

We can provide default values for parameters that have not been passed by the function callers. This is how we can take advantage of default values to write a function that formats a date and writes it to the console:

```
PS> function Format-Date($date, $format = "{0:M/d/yyyy}")
{
    Write-Host ($format -f $date)
}
```

```
PS> Format-Date [datetime]::Today
[datetime]::Today
PS> Format-Date ([datetime]::Today)
9/23/2007
```

As you can see, the default date format is month/day/year. We can override that, say, if we wanted to use a full date and time pattern:

```
PS> Format-Date ([datetime]::Today) "{0:f}"
Sunday, September 23, 2007 12:00 AM
```

We can force function callers to always pass one of the parameters by using a default value that throws an exception. This is how we can make the input date parameter mandatory:

```
PS> function Format-Date($date = $(throw "Date required"), `
    $format = "{0:M/d/yyyy}")
{
    Write-Host ($format -f $date)
}
PS> Format-Date
Date required
At line:1 char:37
+ function Format-Date($date = $(throw <<<< "Date required"), `
```

When I said that we can use exactly the same techniques as we did with script blocks, I meant it. We can even skip declaring parameters in the function definition and declare them inside the function body. The body is just a script block, and it can contain a `param` statement. Here is how our `Format-Date` function looks with the parameters declared inside the script block:

```
PS> function Format-Date
{
    param ($date = $(throw "Date required"), $format = "{0:M/d/yyyy}")
    Write-Host ($format -f $date)
}

PS> Format-Date (Get-Item C:\autoexec.bat).LastAccessTime
11/2/2006
```

Passing Parameters by Reference

Parameters passed to a function transfer data in one direction only. We cannot assign a value to a parameter and expect the original value to change. We can assign new values to parameters, but that will simply create a local variable with the same name. Here is what happens:

```
PS> $name = "John"
PS> function AssignToParam($name)
{
    $name = "Mike"
    Write-Host "inside function: $name"
}
PS> AssignToParam $name
inside function: Mike
PS> Write-Host "outside function: $name"
outside function: John
```

The newly created variable will shadow the one that was originally passed and that will preserve the original value.

How can we change the parameter that was passed to our function? As we saw in the previous chapter, we can traverse the variable scope upwards using `Get-Variable` and `Set-Variable`, and that will definitely work. Let's use that technique to create a function that will swap two variables' values. Warning—ugly code ahead!

```
PS> function Swap($first, $second)
{
    $firstValue = Get-Variable $first -scope 1 -valueOnly
    $secondValue = Get-Variable $second -scope 1 -valueOnly
    Set-Variable $first $secondValue -scope 1
    Set-Variable $second $firstValue -scope 1
}

PS> $a = 4
PS> $b = 5
PS> Swap "a" "b"
PS> $a
5
PS> $b
4
```

I cannot even start to describe how bad I feel for having to pass the variable names instead of values. Unfortunately, providing the value only is not sufficient—we need the name in order to modify the variable. Not only is the preceding code extremely ugly but it might not work in all cases. What if the variables are declared not in the parent scope but in the grandparent one instead? We do not really know if we should pass 1 or 2 or an even greater number as the `-scope` parameter.

Fortunately, there is a better way than looking for variables in all parent scopes. PowerShell provides support for variable references through the `PSReference` object. We can cast any variable to a `PSReference` using the `[ref]` type shorthand, and we will get an object whose `Value` property we can modify. Setting a reference's `Value` property will modify the original variable. This is how we can rewrite our `Swap` function to make it use references:

```
PS> function Swap([ref] $first, [ref] $second)
{
    $tmp = $first.Value
    $first.Value = $second.Value
    $second.Value = $tmp
}

PS> $a = 4
PS> $b = 5
PS> Swap ([ref] $a) ([ref] $b)
PS> $a
5
PS> $b
4
```

Much better! The code is cleaner, and it refers to the variables unambiguously without having to probe parent scopes. Note the use of parentheses at the line that calls `Swap`; we have to explicitly convert the variables to references before passing them to the function.

Returning Values

Just as you saw with script blocks, you can return a value from a function by outputting an object that is not consumed by any operation. Outputting more than one object will result in the function returning a collection. Here is a function that outputs several objects in a loop:

```
PS> function Generate-Numbers($max)
{
    for ($i = 0; $i -lt $max; $i++)
    {
        $i
    }
}
```

```
PS> Generate-Numbers 3
0
1
2
```

We can return a value and exit the function at the same time using the `return` statement. Here is how to use that to create a function that searches for an object in a collection:

```
PS> function Find-Object($needle, $haystack)
{
    foreach ($item in $haystack)
    {
        if ($item -eq $needle)
        {
            return $item
        }
    }
}
```

```
PS> Find-Object 5 (3..8)
5
PS> Find-Object 5 (2..4)
PS>
```

I want to make an important distinction here between outputting objects and writing objects to the console. Outputting objects is the act of passing objects down the output stream. We can do that on purpose, by using the variable name as we did in the `Generate-Numbers` example, or we can simply let a command we call return values without collecting them in a variable. Here is how we can do this with a simple `dir` call:

```
PS> function Get-TextFiles()
{
    dir *.txt
}
```

```
PS> Get-TextFiles
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\PowerShell
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	9/14/2007 12:31 AM	12	digits.txt
-a---	9/13/2007 6:33 AM	11266	largetext.txt
-a---	7/7/2007 11:33 PM	4	other.txt

PowerShell will pass all output objects to the next command down the pipeline.

Writing objects is the explicit conversion of objects to text and sending that text to the console. No values will be passed down the pipeline. This is what the `Write-Host` cmdlet, which we have used in previous examples, does. Outputting and writing objects look almost the same because of the fact that every pipeline implicitly ends with a call to the `Out-Default` cmdlet. That cmdlet processes all input and writes it to the console.

Scoping Rules

Functions, just like script blocks, create a new local scope. Again, the scope inheritance variable visibility rule applies: a function can read all variables declared in its scope and all of its parent scopes. Refer to Chapter 4 for an in-depth discussion of variable scoping.

Being named objects, functions obey scoping rules in a way similar to variables. We can declare a function in any scope, and it will be visible to code in that scope and its child scopes. That means we can nest functions. Here is what nesting looks like:

```
PS> function Outer()
{
    function Inner()
    {
        Write-Host "Inner function"
    }
    Inner
}
```

```
PS> Outer
Inner function
```

The `Inner` function is visible to code inside the `Outer` one only. We will get an error if we try to call it from the outside scope:

```
PS> Inner
The term 'Inner' is not recognized as a cmdlet, function, operable program,
or script file. Verify the term and try again.
At line:1 char:5
+ Inner <<<<
```

Similar to variable assignment, defining a function in a child scope will always shadow functions with the same name in any of the parent scopes. Here is how we can override what a function does for the current scope:

```
PS> function Outer()
{
    function Do-Something()
    {
        Write-Host "Original Do-Something"
    }
    function Inner()
    {
        function Do-Something()
        {
            Write-Host "Overriden Do-Something"
        }
        Do-Something
    }
    Inner
    Do-Something
}
```

```
PS> Outer
Overriden Do-Something
Original Do-Something
```

The `Outer` function defines the `Do-Something` and `Inner` nested functions. The `Inner` function, in turn, defines another `Do-Something` function. That inner definition overrides the `Do-Something` function inherited from the parent scope, so calling the `Inner` function prints “Overriden Do-Something.” That override is valid only for code executing in the `Inner` function’s scope. Once we exit that scope, `Do-Something` refers to its original value.

Unlike `Get-Variable`, the `Get-Command` cmdlet does not support a scope parameter, and there is no way to get to a function from the parent scope once we override it in the current scope.

In most cases, we can prefix a function name with the `global`, `script`, `local`, or `private` scope modifier to help us get to the correct function at the time of execution. Here is how we can declare both a global and a local function having the same names and use the namespace prefixes to distinguish between the two:

```
PS> function global:Do-Something()
{
    Write-Host "Global Do-Something"
}
function InnerScope()
{
    function local:Do-Something()
    {
        Write-Host "Local Do-Something"
    }
}
```

```
    local:Do-Something
    global:Do-Something
}
```

```
PS> InnerScope
Local Do-Something
Global Do-Something
```

You can refer to the previous chapter for in-depth discussion of the scope modifier prefixes.

Note The scope namespace prefixes work in the same way for functions as they do for variables. The syntax is slightly different though: if we are referring to a global variable, we must use a `$global:myVariable` syntax. Dollar signs are used to distinguish variables from other programming language constructs, and they make parsing the language easier for the PowerShell developers. Therefore, namespace prefixes for functions do not use dollar signs, and we have to use a `global:myFunction` syntax to refer to global functions.

When we override local functions at different scope levels, we can use a clever hack to call a function from a parent scope, but doing so will require that we obtain a reference to the function before overriding it in the current scope. Here is how to do that:

```
PS> function Outer()
{
    function Do-Something()
    {
        Write-Host "Original Do-Something"
    }
    function Inner()
    {
        $original = Get-Command Do-Something -type Function
        function Do-Something()
        {
            Write-Host "Overriden Do-Something"
            Write-Host "Calling original function..."
            &$original
        }
        Do-Something
    }
    Inner
}
```

```
PS> Outer
Overriden Do-Something
Calling original function...
Original Do-Something
```

This is still a clumsy solution, because in the real world, we may have to check if a function with the same name already exists before declaring one ourselves, so that we can get the original value. Hopefully, a situation like this one will occur very rarely and can easily be avoided.

Filters

As you already know, most PowerShell scripting is about pipelines. The way to accept pipeline input in a function is to define `begin`, `process`, and `end` sections inside the function. Those sections are executed when the function has something piped to it. Of the three sections, `process` is the only mandatory one—it is executed for each object that has been piped in. As usual, inside the `process` section, we can refer to the current object using the `$_` special variable. The `begin` and `end` sections are executed before and after the pipeline has been processed. Since an example is worth a thousand words, here is a function that accepts a pipeline of files and calculates their total size:

```
PS> function Get-Size
{
    begin
    {
        $total = 0
    }
    process
    {
        Write-Host "processing: $($_.Name)"
        $total += $_.Length
    }
    end
    {
        return $total
    }
}
```

```
PS> dir *.txt | Get-Size
processing: test.txt
processing: test2.txt
processing: test3.txt
processing: test4.txt
34
```

Most often, when dealing with pipeline input, we will need to define only a `process` section. For example, we can filter a collection of processes and display only the ones started less than five minutes ago. To do that, we define a `Get-RecentlyStarted` function that returns the current pipeline object from its `process` section if its `StartTime` property is less than five minutes ago:

```

PS> function Get-RecentlyStarted
{
    process
    {
        $start = $_.StartTime
        if ($start -ne $null)
        {
            $now = [datetime]::Now
            $diff = $now - $start
            if ($diff.TotalMinutes -lt 5)
            {
                return $_
            }
        }
    }
}

```

```
PS> Get-Process | Get-RecentlyStarted
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
187	11	22336	37808	121	5.38	3316	firefox
48	3	1236	7624	59	0.13	912	notepad

PowerShell does not allow a function to contain both arbitrary statements that get executed when the function is called and the pipeline-related `begin`, `process`, and `end` sections. There is a deep semantic difference between collecting all input and passing it to the function for immediate processing and calling a function once the first item becomes available then calling it for the second item and so on. That is why the two modes of function operation are deemed incompatible.

The PowerShell language designers are fully aware of the profound difference between regular and pipeline-processing functions. The pipeline function mode is often used to filter a collection, and that has led to implementing a shorthand notation that helps in defining filter functions. We can do that using the `filter` keyword. Let's now rewrite the previous example as a filter:

```

PS> filter Get-RecentlyStarted
{
    $start = $_.StartTime
    if ($start -ne $null)
    {
        $now = [datetime]::Now
        $diff = $now - $start
        if ($diff.TotalMinutes -lt 5)
        {
            return $_
        }
    }
}

```

```
PS> Get-Process | Get-RecentlyStarted
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
188	11	22376	37780	121	4.63	3244	firefox
48	3	1236	7616	59	0.14	2468	notepad

As you can see, using filters instead of functions unclutters our code by eliminating a nesting level and makes it much more readable.

We can get a closer look at filters again using `Get-Command`. According to PowerShell, filters are functions, because asking for commands of the `Function` and `Filter` type will both return our filter:

```
PS> Get-Command Get-RecentlyStarted -type Function
```

CommandType	Name	Definition
Function	Get-RecentlyStarted	process {...

```
PS> Get-Command Get-RecentlyStarted -type Filter
```

CommandType	Name	Definition
Function	Get-RecentlyStarted	process {...

And here is the hidden process section that each filter has:

```
PS> (Get-Command Get-RecentlyStarted -type Filter).ScriptBlock
process {
$start = $_.StartTime
  if ($start -ne $null)
  {
    $now = [datetime]::Now
    $diff = $now - $start
    if ($diff.TotalMinutes -lt 5)
    {
      return $_
    }
  }
}
```

The preceding example should prove that filters are just syntactic sugar sprinkled on top of functions.

FUNCTIONS' AND FILTERS' INTERNAL REPRESENTATION

Functions that work with the pipeline and filters are so similar that you may think that internally they are represented by the same objects. This is not quite the case: although a function with a single `process` block is semantically equivalent to a filter, it is still represented by an internal `FunctionInfo` object, while a filter is represented by a `FilterInfo` one.

As an example, take the following function that filters out all odd numbers from the pipeline, effectively returning even numbers only:

```
PS> function Even-Function
{
    process
    {
        if ($_ % 2 -eq 0)
        {
            $_
        }
    }
}
```

Now, let's take a look at an implementation of the same logic that uses a filter:

```
PS> filter Even-Filter
{
    if ($_ % 2 -eq 0)
    {
        $_
    }
}
```

Although both `Even-Function` and `Even-Filter` as defined previously are returned when we ask for commands of the `Function` type, we get a different object for each of them:

```
PS> Get-Command Even-Function -CommandType Function
```

CommandType	Name	Definition
Function	Even-Function	process {...

```
PS> Get-Command Even-Filter -CommandType Function
```

CommandType	Name	Definition
Filter	Even-Filter	process {...

Another way to distinguish between a function and a filter at runtime is to check the `IsFilter` property of the inner `ScriptBlock` object. Here is how:

```
PS> $function = (Get-Command Even-Function -CommandType Function)
PS> $function.ScriptBlock.IsFilter
False
PS> $function = (Get-Command Even-Filter -CommandType Function)
PS> $function.ScriptBlock.IsFilter
True
```

To reiterate, those two function objects work in absolutely the same way. Being able to distinguish between the two might come in handy only in cases when we are working on reflective scripts printing diagnostic information about the environment or when debugging a really wicked problem.

Functions and Script Blocks

Combining functions and script blocks results in some very powerful programming techniques. By combining functions' ability to create new commands with the dynamic nature of script blocks, we can get very close to the built-in PowerShell language syntax. Calling a function that accepts a single script block closely resembles using the `if`, `while`, or `switch` statements. That makes our code look as if it builds and uses new language constructs that makes it easier to read and understand. In addition to the cool syntax, script blocks make a perfect tool for encapsulating actions or strategies that can be executed by a function as it traverses various objects.

Implementing New Control Structures

I do not remember anymore how many times I have written complex loops that require a nontrivial exit condition. Those are often coded as an endless loop and contain a `break` statement that exits the loop. The norm for writing such loops is to use a `while` or a `for` loop like this:

```
PS> while ($true)
{
    Write-Host "Looping..."
    #something triggers the exit condition
    break
}

Looping...
PS> for (;;)
{
    Write-Host "Looping..."
    #something triggers the exit condition
    break
}

Looping...
```

To save some keystrokes, we can introduce a new statement, `loop`, that repeatedly executes a block. It will use a hidden `while` loop to execute the block many times:

```
PS> function loop($code)
{
    while ($true)
    {
        &$code
    }
}

PS> loop {
    write-host "Looping..."
    break;
}
Looping...
```

See how it looks almost like a regular control-flow statement? The only difference is that we have to place the opening brace on the same line or use a line continuation escape using the backtick (```) symbol, so the shell evaluates the block and passes it to the `loop` function. The fact that our script block is executed in a `while` loop allows us to use loop-related control flow statements like `break` and `continue`. As you can see in the preceding example, they still work.

Another convenient extension we can define is an operation that will silently ignore all errors. The following code defines a function `ignore` that executes a script block once, catches any exceptions thrown by the block, and silently continues:

```
PS> function ignore($code)
{
    trap
    {
        continue
    }
    &$code
}

PS> $numbers = 2..0
PS> foreach ($number in $numbers)
{
    ignore {
        5 / $number
    }
}

2.5
5
```

Our function just sets up an exception trap that will do nothing when an error occurs and then execute the script block that has been provided. We use the `ignore` function to perform several division operations and ignore all errors: we see two result values because the third

iteration divided by zero, which raised an exception that we ignored. We will discuss exception handling and debugging in detail in Chapter 9.

Another useful extension that we can build can help us get diagnostic messages about a command. Often, to ensure that a piece of code has executed, we have to add `Write-Host` statements both before and after it: If the statement before the code executes, our code is being executed. If the statement after it is executed, we are sure that our code does not raise any errors. We can automate scattering `Write-Host` statements in our code by implementing a trace function that takes two parameters: the current action description and the action itself. It writes to the console a log message containing the action description we provided both before and after executing the action. Here is the code:

```
PS> function trace($description, $code)
{
    Write-Host "Before: $description"
    &$code
    Write-Host "After: $description"
}

PS> trace "dir" {
    dir *.txt
}
```

Before: dir

Directory: Microsoft.PowerShell.Core\FileSystem::C:\PowerShell

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	9/14/2007 12:31 AM	12	digits.txt
-a---	9/13/2007 6:33 AM	11266	largetext.txt

After: dir

```
PS> trace "division" {
    $denominator = 0
    3 / $denominator
}
```

Before: division
 Attempted to divide by zero.
 At line:3 char:8
 + 3 / <<<< \$denominator
 After: division

Having `trace` in our toolbox makes it easy to surround any statement with a script block whose execution will get logged to the console.

Combining our extensions is extremely powerful. Here is how we can trace a division by zero while ignoring all errors by nesting an ignore block inside a trace one:

```
PS> trace "division" {
    ignore {
        $denominator = 0
        3 / $denominator
    }
}
```

Before: division

After: division

We can use techniques similar to trace to write to files, measure and profile function execution times, and so on.

Script Blocks as Strategies

PowerShell does an excellent job of traversing collections and operating with pipelines. It excels at manipulating flat collections. Quite often, we have to deal with data that is organized in a complex way. Linear traversal of collections often is inadequate for the problem at hand. Suppose we have to walk the file system or an XML document tree. Those two objects have a hierarchical structure that is best traversed recursively. Since PowerShell does not provide primitives for walking over trees, we will have to write them ourselves. Suppose that we have to go through a large XML document and extract information from various nodes, which we want to quickly find according to some condition. Suppose we have this HTML-like data stored in a file called `data.xml` residing the current folder:

```
<body>
some text and an unordered list
  <ul>
    <li><a href="doc1.xml">doc1</a></li>
    <li><a href="doc2.xml">doc2</a></li>
    <li>EMPTY</li>
  </ul>
</body>
```

We can rapidly load it in an XML document with the following command:

```
PS> $doc = [xml](Get-Content data.xml)
PS> $doc
```

body

Body

Now, we want to obtain the first `` element that contains an `<a>` tag. We create a recursive function that starts with the `<body>` element and traverses its children, then the children of their children, and so on until it finds the elements we are looking for. Here is the code that does that:

```

PS> function Find-LiAnchor($root)
{
    $name = $root.PSBase.Name
    if ($name -eq "li" -and $root.a -ne $null)
    {
        return $root
    }
    else
    {
        foreach ($child in $root.PSBase.ChildNodes)
        {
            $result = Find-LiAnchor -root $child -strategy $strategy
            if ($result)
            {
                return $result
            }
        }
    }
}

```

```

PS> Find-LiAnchor($doc.body)

```

```

a
-
a

```

The function returns the first XML element that it finds. We exit the recursion by checking if a child has already been identified as matching our search criteria. Note that we have to use the `PSBase` property to get to the raw, unadapted `XmlElement` object and its `Name` property. The XML type adapter prevents us from getting the tag name by getting the `Name` property directly.

Everything looks fine with our approach until we face our next task: finding the first `` element without a nested `<a>` inside. We would not want to copy and paste the recursive function and change the condition. Pulling the condition into a script block is the best option, as that will allow us to write a generic `Find-Node` function. That script block is our search strategy. It will be applied to all XML elements, and it should return `$true` for the one we are after, and `$false` otherwise. Here is our modified code with the search strategy encapsulated in a script block:

```

PS> function Find-Node($root, $strategy)
{
    $foundNode = &$strategy -node $root
    if ($foundNode)
    {
        return $root
    }
    else
    {

```

```

        foreach ($child in $root.PSBase.ChildNodes)
        {
            $result = Find-Node -root $child -strategy $strategy
            if ($result)
            {
                return $result
            }
        }
    }
}

```

```

PS> $emptyLI = Find-Node -root $doc.body -strategy {
    param($node)
    return ($node.PSBase.Name -eq "li") -and ($node.a -eq $null)
}

```

```

PS> Write-Host $emptyLI
li

```

With that `Find-Node` implementation in place, our element searches become almost trivial: all we have to do is define our search condition with a one- or two-line block. Let's see how we can find the first `<a>` element that has a nonnull `href` attribute:

```

PS> $anchor = Find-Node -root $doc.body -strategy {
    param($node)
    return ($node.PSBase.Name -eq "a") -and `
        ($node.PSBase.GetAttribute("href") -ne $null)
}

```

```

PS> Write-Host $anchor.href
doc1.xml

```

Again, we are using `PSBase` to get to the raw `Name` property and the `GetAttribute()` method. We need `GetAttribute()` as a way to check if an element contains a specific attribute. We could have used a `$node.href -ne $null` condition, but that would return true in cases where we have a child `<href>` element as the XML type adapter converts both attributes and nested elements to properties.

Summary

In this chapter, we continued our journey through the abstraction-building mechanisms that PowerShell offers. Functions are the most common way to extend the built-in shell functionality, and mastering them early on is vital. The next sections deal with complex scenarios, and the only way to cope with those is to break a problem up into smaller manageable chunks and solve them one by one. This is the “divide and conquer” approach to problem solving. We will solve each subproblem by writing a function, and in the end, we will glue several functions together to get to the ultimate solution.



Command Aliases

Command aliases are a powerful addition to any shell scripter's toolbox. For decades, UNIX shell users have been able to alias a command to something shorter and friendlier. Unfortunately, the DOS-based shells, `command.com` and `cmd.exe`, lacked that feature. In practice, external tools like `doskey.exe` allow users to define an alias for a command, but they are not built into the shell, and users have to learn their quirky syntax. PowerShell finally fills that gap in Windows shells and provides built-in support for command aliases.

Aliases can be extremely helpful in relieving the strain on your wrists by saving enormous amounts of typing. PowerShell provides shorter alias names for its verbose cmdlet names out of the box and encourages users to define their own aliases, so that they can be as productive as possible.

Aliases have a dark side too—we can get into all sorts of trouble if we do not use them wisely, so I will show you how to detect and avoid several alias-related problems. Aliases are a wonderful migration tool; we can use them to provide commands that users coming from another environment are accustomed to seeing. PowerShell does exactly that by providing aliases to some of the built-in commands that will make `cmd.exe` and UNIX shell users feel at home. I will briefly outline those aliases and focus on things that are different in PowerShell with regard to `cmd.exe` and UNIX shells.

Working with Aliases

PowerShell offers a host of useful commands—quite a few of them come with the default installation; Version 1.0 comes with more than a hundred. Here is how to get the exact number of cmdlets registered on your system:

```
PS> (get-command -type Cmdlet).Length  
129
```

129 is quite a large number! The PowerShell designers have imposed a lot of rules and guidelines on cmdlet authors, so that names and parameters are consistent across the system and users learn easily. Even with cmdlet naming conventions and all the similarities across commands, learning more than a hundred of them can be a daunting task. What do you do if you cannot trick your mind into remembering some of the names? That is right; you cheat and set up an alias for that command so that you get a name that is easier to recall.

Creating Aliases

An alias is a way to associate a name with an existing command. If you are like me, you will soon get tired of writing the `Write-Host` cmdlet name and will pick something shorter. I use that cmdlet when I want to log something to the console for diagnostics purposes. How do we avoid typing that full name all the time? Simple—define a log alias that refers to `Write-Host`. We do that by calling the `New-Alias` cmdlet:

```
PS> New-Alias log Write-Host
PS>
```

We can now use that alias in our code:

```
PS> log "Operation completed successfully."
Operation completed successfully.
```

What happens if somebody has already defined that alias for us or even if we did so and forgot about it? `New-Alias` will raise an error if the alias already exists:

```
PS> New-Alias log Get-ChildItem
New-Alias : Alias not allowed because an alias with the name 'log' already exists.
At line:1 char:10
+ New-Alias <<<< log Get-ChildItem
```

We can check the list of aliases to verify if the alias name we want to use is available. We have three ways to do that: the `Get-Alias` and `Get-Command` cmdlets and the `Alias` provider.

Get-Alias

This cmdlet returns either all aliases on the system or just the ones we need according to the filter specified. Here is how to get all aliases that start with the “l” character:

```
PS> Get-Alias l*
```

CommandType	Name	Definition
-----	----	-----
Alias	lp	Out-Printer
Alias	ls	Get-ChildItem
Alias	log	Write-Host

Of course, we can use the full name and call that in an if statement:

```
PS> if (Get-Alias log)
{
    log "Alias exists"
}
```

```
Alias exists
```

We are a bit bold here when calling that alias, because we assume that if the alias exists, it is our own alias, or even if it is not, it does something similar to logging to the console. We do

that with all hope that people defining aliases that delete files and naming them `log` will spend a painful eternity writing Perl scripts.

Get-Command

Aliases are commands in their own right, and the `Get-Command` cmdlet can retrieve them. We just need to set the `CommandType` parameter to `Alias` or provide a filter that will narrow our results to something manageable. Here is how to get our `l*` aliases:

```
PS> Get-Command -CommandType Alias l*
```

CommandType	Name	Definition
-----	----	-----
Alias	log	Write-Host
Alias	lp	Out-Printer
Alias	ls	Get-ChildItem

Note that we got the same output as we did in the `Get-Alias` example. Both cmdlets have a similar implementation in that respect, and both return `AliasInfo` objects that describe our aliases.

The Alias Provider

By default, the `Alias` provider maps the `alias:` drive, which we can query similar to a file system. Here is how to navigate there and get the `l*` aliases:

```
PS> cd alias:
PS> dir l*
```

CommandType	Name	Definition
-----	----	-----
Alias	lp	Out-Printer
Alias	ls	Get-ChildItem
Alias	log	Write-Host

Again, we get a collection of `AliasInfo` objects, because the `Alias` provider returns items of that type. Having the provider on our side, we can turn the check for an alias's existence into a simple item existence check by using the `Test-Path` cmdlet that we would normally use to verify if a file or a folder exists. Here is how to rewrite the preceding `if` statement using `Test-Path`:

```
PS> if (Test-Path alias:\log)
{
    log "Alias exists"
}
```

```
Alias exists
```

For an in-depth discussion of providers and provider items, see the next chapter.

Modifying Aliases

Adding aliases and verifying if an alias exists is only a part of the big picture. Quite often, we want to modify an alias or explicitly overwrite an existing alias with our own value. As you already saw, `New-Alias` will raise an error if the alias already exists. We can tell it that we want to force the creation of an alias by passing the `Force` switch parameter. That will suppress the error, silently delete the old alias, and create a new one in its place. This is how we can update our `log` alias and make it use `Write-Verbose` instead of `Write-Host`:

```
PS> Get-Alias log
```

CommandType	Name	Definition
-----	----	-----
Alias	log	Write-Host

```
PS> New-Alias log Write-Verbose -Force
```

```
PS> Get-Alias log
```

CommandType	Name	Definition
-----	----	-----
Alias	log	Write-Verbose

```
PS> $VerbosePreference = "Continue"
```

```
PS> log "This is a test"
```

```
VERBOSE: This is a test
```

Note that we need to set the `$VerbosePreference` global variable to `Continue` so that we see our message at the console and execution is not stopped. You can learn more about script debugging techniques, such as logging, tracing, and stepping through script code, in Chapter 9.

Instead of forcing us to force new aliases, PowerShell provides an alternative way of modifying an alias's definition: the `Set-Alias` cmdlet. It works just like `New-Alias -Force` and is more concise. Here is how to change the `log` alias back to `Write-Host`:

```
PS> Set-Alias log Write-Host
```

```
PS> Get-Alias log
```

CommandType	Name	Definition
-----	----	-----
Alias	log	Write-Host

```
PS> log "Write-Host is back"
```

```
Write-Host is back
```

There is another way to do this operation: via the `Alias` provider. Because an alias is an item, just like a file, we can set that item to something new using the `Set-Item` cmdlet. Here it is in action:

```
PS> cd alias:
PS> Set-Item log Write-Verbose
PS> dir log
```

CommandType	Name	Definition
Alias	log	Write-Verbose

An alias's definition is its content, and we can modify an alias by calling `Set-Content`. Now, let's assume we can't figure out what `log` should really do and decide to switch back to `Write-Host`:

```
PS> cd alias:
PS> Set-Content log Write-Host
PS> dir log
```

CommandType	Name	Definition
Alias	log	Write-Host

It might look strange, but PowerShell does not provide a cmdlet that will delete or unset an alias for us. We are on our own here. To get rid of an unneeded alias, we have to use the `Alias` provider and call the `Remove-Item` cmdlet (conveniently aliased to `del`). Let's now get rid of the `log` alias:

```
PS> cd alias:
PS> del log
PS> Get-Alias log
Get-Alias : Cannot find alias because alias 'log' does not exist.
At line:1 char:10
+ Get-Alias <<<< log
```

Our trusty alias now rests in peace, and PowerShell complains if we look for it using `Get-Alias`.

Exporting and Importing Aliases

Imagine the following situation: You have been customizing your shell with tons of useful aliases. You are a productivity demigod, and you do all your scripting tasks in record time. A coworker of yours is having some trouble with his shell and asks for help. You go to his machine, fire up PowerShell, and start typing. You get stuck all the time and your productivity plummets, because your muscle memory keeps making your fingers type those alias names that you are used to and those command shortcuts are not available on your coworker's machine. Does that mean you should avoid customizing your environment, so that you do not feel lost when you use another machine? Of course not! That would chain us to lower productivity levels forever. The solution is to export your aliases and import them on the machine you are working on at the moment. This way, you get increased productivity and customizability along with high portability for your skills.

To export your aliases, you have to invoke the `Export-Alias` cmdlet. At the very minimum, you can just provide a file name where the new aliases will get stored. The file is a plain text file and looks like this:

```
PS> Export-Alias aliases.txt
PS> type aliases.txt
# Alias File
# Exported by : Hristo
# Date/Time : Wednesday, October 10, 2007 11:07:02 PM
# Machine : NULL
"ac","Add-Content","", "ReadOnly, AllScope"
"asnp","Add-PSSnapin","", "ReadOnly, AllScope"
"clc","Clear-Content","", "ReadOnly, AllScope"
"cli","Clear-Item","", "ReadOnly, AllScope"
"clp","Clear-ItemProperty","", "ReadOnly, AllScope"
"clv","Clear-Variable","", "ReadOnly, AllScope"
"cp","Copy-Item","", "ReadOnly, AllScope"
"cpp","Copy-ItemProperty","", "ReadOnly, AllScope"
"cvpa","Convert-Path","", "ReadOnly, AllScope"
"diff","Compare-Object","", "ReadOnly, AllScope"
...
```

You can get that file to the target machine and then use `Import-Alias` to get your aliases installed in the current shell session:

```
PS> Import-Alias aliases.txt
Import-Alias : Alias not allowed because an alias with the name 'ac' a
lready exists.
At line:1 char:13
+ Import-Alias <<<< aliases.txt
Import-Alias : Alias not allowed because an alias with the name 'asnp'
already exists.
At line:1 char:13
+ Import-Alias <<<< aliases.txt
Import-Alias : Alias not allowed because an alias with the name 'clc'
already exists.
At line:1 char:13
+ Import-Alias <<<< aliases.txt
...
```

Do not get scared by all the error messages. They are caused by the default `Export-Alias` behavior—by default, the cmdlet will export *all* aliases on the system, even the built-in ones, so `Import-Alias` will generate nonfatal errors when it tries to import them. That is not a problem, as the built-in alias definitions are the same on every machine anyway, and we do not care if we redefine those or not.

FORCING BUILT-IN ALIASES REDEFINITION FOR THE PARANOID

Some may even argue that we are better off redefining the default aliases for a new shell session if we are working on another machine, as somebody might have changed them. Reverting to the default behavior is a good idea, as it will save us hair pulling frustration in trying to figure out some awkward problem. There are security implications too—you would not want to execute malicious code that has been aliased with a name that makes it look like a built-in command.

You can restore defaults by forcing the alias import process to overwrite existing aliases. To do that, pass the `Force` parameter to `Import-Alias`:

```
PS> Import-Alias aliases.txt -Force
PS>
```

Do not worry that you might destroy somebody's legitimate alias—the newly imported aliases will live only as long as the shell session does; they are not persisted for future shell sessions. See the “Persisting Aliases for All Shell Sessions” sidebar for details on enabling aliases for all shell sessions.

Anyway, we can either ignore the import errors by providing the correct `ErrorAction` parameter, or edit the text file and remove the built-in aliases. Here is how to ignore the errors:

```
PS> Import-Alias aliases.txt -ErrorAction SilentlyContinue
PS>
```

Editing the text file can get a bit tough, because we may have to find and delete a lot of content. It is always best if we know what our aliases look like and export only those by providing a filter. For example, here is how we can export all aliases that start with `log`:

```
PS> Set-Alias log-host Write-Host
PS> Set-Alias log-verbose Write-Verbose
PS> Export-Alias log-aliases.txt log*
PS> type log-aliases.txt
# Alias File
# Exported by : Hristo
# Date/Time : Wednesday, October 10, 2007 11:20:27 PM
# Machine : NULL
"log-host","Write-Host","", "None"
"log-verbose","Write-Verbose","", "None"
```

We can even do multiple exports to the same file by appending the contents. Here is how to add the `trace*` aliases to our file:

```
PS> Set-Alias trace-host Write-Host
PS> Export-Alias log-aliases.txt trace* -append
PS> type log-aliases.txt
```

```
# Alias File
# Exported by : Hristo
# Date/Time : Wednesday, October 10, 2007 11:20:27 PM
# Machine : NULL
"log-host","Write-Host","", "None"
"log-verbose","Write-Verbose","", "None"
# Alias File
# Exported by : Hristo
# Date/Time : Wednesday, October 10, 2007 11:22:46 PM
# Machine : NULL
"trace-host","Write-Host","", "None"
```

Note that the file headers get inserted twice, but that is not a problem, as the import process ignores them.

Suppose that you want to get not a text file but a script file that you can execute and use to add aliases. Doing that is as simple as passing the `-as` parameter:

```
PS> Export-Alias log-alias.ps1 log* -as Script
PS> type log-alias.ps1
# Alias File
# Exported by : Hristo
# Date/Time : Wednesday, October 10, 2007 11:27:46 PM
# Machine : NULL
set-alias -Name:"log-host" -Value:"Write-Host" -Description:"" -Option
:"None"
set-alias -Name:"log-verbose" -Value:"Write-Verbose" -Description:"" -
Option:"None"
```

Note how the script contains the actual calls to `Set-Alias` that will define the alias on the target machine. You can also export aliases to comma-separated files. That might come in handy when creating documentation, as you can process comma-separated data quite easily using Microsoft Excel and other table-oriented tools. Here is how to get a CSV file with your alias definitions:

```
PS> Export-Alias log-alias.csv log* -as Csv
PS> type log-alias.csv
# Alias File
# Exported by : Hristo
# Date/Time : Wednesday, October 10, 2007 11:29:18 PM
# Machine : NULL
"log-host","Write-Host","", "None"
"log-verbose","Write-Verbose","", "None"
```

PERSISTING ALIASES FOR ALL SHELL SESSIONS

Any alias that you define is valid throughout only the current shell session's lifetime. Aliases are just like variables and functions; they are transient objects living in the current process's memory. You have two options for saving aliases so that they become permanently available in all shell sessions:

- Add the Set-Alias calls that define your aliases to your shell's profile. You can learn more about configuring your shell through the profile script in Chapter 11. This is the easiest option, as you only need to copy a bunch of code lines from your console window and paste them in the profile script. The downside is that that script gets messy and hard to maintain if you have many aliases defined.
- Export the aliases you need from your shell session after you finish defining them. Edit the file, and remove the unneeded aliases. Place that file in a central location on your computer, and add an Import-Alias command to your profile script that will import the aliases there. Probably the best place to store the alias file is next to your profile script. That place will be the first anyone (you included) will look for when trying to modify the shell configuration.

Aliasing Tips, Techniques, and Pitfalls

When it comes to shell scripting, aliases are not, strictly speaking, assets—they can be liabilities too. They shine at saving typing, but they can really hurt readability, because they force the reader to remember what each alias points to. Compare the following snippet:

```
Get-ChildItem *.txt | Where-Object { $_.Size -gt 5KB } | `
    ForEach-Object { $_.Name }
```

to this one:

```
gci *.txt | ? { $_.Size -gt 5KB } | % { $_.Name }
```

Which one is easier to understand? The first one reads almost like English, while the second may as well be Klingon—even seasoned PowerShell users will have to stop and remember that `?` is an alias for `Where-Object` and `%` for `ForEach-Object`. A good rule of thumb that I like to follow is to use shorter aliases when I am typing on the console. When creating a reusable script, I always try to provide the longer readable names. This makes it easier on the poor person who will read and modify the script next. If you need an additional reason to adhere to that rule, keep in mind that most likely that that poor person will be you.

Name Clashes

The most important part of an alias is its name, so we have to be extra careful when we name an alias, because it can and will shadow all other commands. For example, let's create an alias and name it `Get-Command`:

```
PS> Set-Alias Get-Command dir
PS> Get-Command
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	9/5/2007 10:37 PM		Code Snippets
d----	8/18/2007 4:32 PM		Downloads
...			

Suddenly, we realize that there is no way for us to call the built-in `Get-Command` cmdlet. Our alias shadows that, and typing `Get-Command` invokes the aliased `dir` command instead of the cmdlet. This is by design. When you type a command at the command line, these are the places that PowerShell looks for commands, in order of priority:

1. *Aliases*: The shell will first try to find an alias. If it does, it executes the alias and ends the search.
2. *Functions*: Next, the shell will try the function definitions.
3. *Cmdlets*: After that, it will go over the registered cmdlets.
4. *Executables*: The shell then checks the executable files in the current directory and directories in the system PATH.
5. *Scripts*: Then, the shell tries PowerShell script files that have a `.ps1` extension.
6. *Normal files*: As a last resort, the shell tries to launch a file with its registered application.

Things get really hectic if an alias has shadowed one of the commands used everywhere. For example, aliasing `Get-ChildItems` to something stupid like `Write-Host` will completely break the shell:

```
PS> Set-Alias Get-ChildItem Write-Host
PS> dir c:\
c:\
```

As you can see, the `dir` alias that points to `Get-ChildItem` is affected too; it too calls `Write-Host`. Internal operations like the built-in tab completion will also start breaking; obviously, that feature uses `Get-ChildItem` under the hood. In short, do not try this at home.

Unfortunately, those types of name clashes can happen unintentionally. It is easy to set an alias, forget about it, set a conflicting one, and then run a command that breaks. The only way to detect that this has happened is to use `Get-Alias` to see if the command has been aliased to something that we do not expect:


```
PS> Get-Alias Get-ChildItem
```

CommandType	Name	Definition
-----	----	-----
Alias	Get-ChildItem	Write-Host

Somebody could unleash his evil genius and alias `Get-Alias` and `Get-Command`. Luckily, as you saw previously, we can get alias information through the `alias:` drive. This is our last stop:

```
PS> Set-Alias Get-Alias Write-Host
PS> Set-Alias Get-ChildItem Write-Host
PS> dir alias:\Get-Alias
alias:\Get-Alias
PS> Remove-Item alias:\Get-ChildItem
PS> Remove-Item alias:\Get-Alias
```

Note how the `dir` command just echoes its input, because it was aliased to `Write-Host`. As you can see, we can remove the offending aliases using `Remove-Item` and restore our shell's normal operation. I hope neither you nor I have to deal with situations where the `alias:` drive has been unmapped and `New-PSDrive` has been aliased to something useless. That would require restarting the shell so that the built-in aliases and commands get restored to a sane state.

Complex Aliases

Aliases provide a mapping from one command name to another. Sometimes, we might want to alias a complex command and even massage parameters before execution. Unfortunately, PowerShell does not support that. If we want to define a `dir-recurse` alias that lists all files recursively we cannot do this:

```
PS> Set-Alias dir-recurse "dir -recurse"
PS> dir-recurse
Cannot resolve alias 'dir-recurse' because it refers to term 'dir -recurse', which is not recognized as a cmdlet, function, operable program, or script file. Verify the term and try again.
At line:1 char:11
+ dir-recurse <<<<
```

Notice that `Set-Alias` did not raise an error, and we got the error when we tried to call the alias? Fortunately, we can still save some keystrokes, but we need to clear the broken alias first. We can then define a function that calls the `dir` command, and passes the `-recurse` parameter for us:

```
PS> del Alias:\dir-recurse
PS> function dir-recurse
{
    dir -recurse
}
```

```
PS> dir-recurse
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\
```

Mode	LastWriteTime	Length	Name
d----	9/5/2007 10:37 PM		Code Snippets
d----	8/18/2007 4:32 PM		Downloads
d----	6/29/2007 4:10 PM		inetpub
...			

We can even afford to use names that shadow built-in cmdlet names, because functions are searched for matches when looking for an executable command before searching cmdlets. To make things even more fun, we can alias functions. Here is how to refer to our `dir-recurse` function by the `dirr` name:

```
PS> Set-Alias dirr dir-recurse
PS> dirr
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\
```

Mode	LastWriteTime	Length	Name
d----	9/5/2007 10:37 PM		Code Snippets
d----	8/18/2007 4:32 PM		Downloads
d----	6/29/2007 4:10 PM		inetpub

Removing Broken Aliases

Aliases come and go, but their definitions remain. The function or script command that an alias refers to may disappear. And `Set-Alias` and `New-Alias` do not offer any form of validation for their command definition when they create an alias. Here is how we can define a completely wrong alias:

```
PS> Set-Alias wrong-alias something-completely-broken
PS> wrong-alias
Cannot resolve alias 'wrong-alias' because it refers to term 'somethin
g-completely-broken', which is not recognized as a cmdlet, function, o
perable program, or script file. Verify the term and try again.
At line:1 char:11
+ wrong-alias <<<<
```

We can define the alias, but we get an error only when we try to invoke it. With time, we can accumulate a big list of broken aliases. We can find those by checking their `Definition` properties and trying to resolve their values as a command. If we get an error, we have a broken alias. Here is how to test that for one command:

```
PS> Get-Command (Get-Alias wrong-alias).Definition
Get-Command : The term 'something-completely-broken' is not recognized
as a cmdlet, function, operable program, or script file. Verify the t
erm and try again.
At line:1 char:12
+ Get-Command <<<< (Get-Alias wrong-alias).Definition
```

We get an error because “something-completely-broken” is, well, completely broken. Let’s now do that for all aliases. We will get only those that return a command for their definition setting:

```
PS> Get-Alias | Where { !(Get-Command $_.Definition `
    -ErrorAction SilentlyContinue) }
```

CommandType	Name	Definition
-----	----	-----
Alias	wrong-alias	something-completely-br...

We can bring this a step further and just delete the matching aliases by extending the pipe with a `ForEach-Object` call that deletes the alias:

```
PS> Get-Alias | Where { !(Get-Command $_.Definition `
    -ErrorAction SilentlyContinue) } | ForEach-Object `
    { Remove-Item "alias:\($_.Name)" }
```

```
PS> Get-Alias wrong*
PS>
```

We need some string operations to get to the correct alias path in the `alias:` drive. Now that we are done with that, we have no more stale aliases on our system.

Built-in Aliases

As I already mentioned, PowerShell comes with a host of aliases right out of the box. Doing a quick check gets us a staggering number:

```
PS> (Get-Alias).Length
101
```

101 built-in aliases! Now that is what I call a lot. We can split those into three logical groups:

- *CMD.EXE compatibility aliases*: A group of aliases aimed at providing similar, if not the same, commands experienced CMD.EXE users know
- *UNIX shell compatibility aliases*: Aliases trying to emulate commands present in UNIX shells like bash
- *Convenience aliases*: Shortcuts for most commonly used commands that can make your life easier

cmd.exe Look-alikes

You have seen already the liberal use of `dir`, `cd`, `copy`, and so on in our examples so far. Those were not the real cmdlet names, just convenient aliases to `Get-ChildItem`, `Set-Location`, and `Copy-Item`, respectively. Table 6-1 provides a full list of the `cmd.exe` compatibility aliases.

Table 6-1. *Built-in PowerShell Aliases that Emulate Commands from cmd.exe*

Name	Definition
<code>cls</code>	<code>Clear-Host</code>
<code>copy</code>	<code>Copy-Item</code>
<code>dir</code>	<code>Get-ChildItem</code>
<code>type</code>	<code>Get-Content</code>
<code>move</code>	<code>Move-Item</code>
<code>popd</code>	<code>Pop-Location</code>
<code>pushd</code>	<code>Push-Location</code>
<code>erase</code>	<code>Remove-Item</code>
<code>rd</code>	<code>Remove-Item</code>
<code>rmdir</code>	<code>Remove-Item</code>
<code>del</code>	<code>Remove-Item</code>
<code>ren</code>	<code>Rename-Item</code>
<code>chdir</code>	<code>Set-Location</code>
<code>cd</code>	<code>Set-Location</code>
<code>set</code>	<code>Set-Variable</code>
<code>echo</code>	<code>Write-Output</code>

Some of those commands behave differently than their `cmd.exe` counterparts and are worth mentioning. The `dir` command, for example, does not support the familiar `/s` switch that recursively gets all files. You will have to use the `-Recurse`, `-rec`, or just `-r` switch. In addition, it has no support for sorting files in any way—you have to pipe the results through `Sort-Object` to do that. We are not forced to make the distinction between deleting a file or a folder: see that `erase`, `delete`, `rd`, and `rmdir` are all aliased to the same command, `Remove-Item`. After all folders, just like files, are items coming out of the `FileSystem` provider. Note how `Set-Variable` has been aliased to `set`. This is different than the `cmd.exe` `set myvar=something` syntax, as you will have to use `set myvar something` in PowerShell. Be extra careful, because the first syntax does not issue an error or a warning. It just does nothing, as shown in this example:

```
PS> set myvar=something
PS> $myvar
PS> set myvar something
PS> $myvar
something
```

Veterans in batch file scripting may have the %myvar% syntax burned into their brains; this is how `cmd.exe` refers to a variable and gets its contents. Take a look at this example, run from within `cmd.exe`:

```
C:\>SET NAME=John

C:\>echo "Hello, %NAME%"
"Hello John"
```

Forget about that syntax! Variables have to be prefixed with a dollar sign in PowerShell. Here is the preceding example in PowerShell:

```
PS> set NAME John
PS> echo "Hello, $NAME"
Hello John
```

Still, this code looks clumsy and not quite in the spirit of PowerShell. It is best written as follows:

```
PS> $name = "John"
PS> echo "Hello, $name"
Hello, John
```

Other than the different variable declaration, assignment, and reference syntax PowerShell is remarkably similar to `cmd.exe`. Seasoned command prompt users will find themselves right at home after their first commands.

UNIX Look-alikes

Long time bash users coming from a UNIX background will not be disappointed with PowerShell. The UNIX-like aliases are listed in Table 6-2.

Table 6-2. Built-in PowerShell Aliases Emulating UNIX Commands

Name	Definition
<code>clear</code>	<code>Clear-Host</code>
<code>diff</code>	<code>Compare-Object</code>
<code>cp</code>	<code>Copy-Item</code>
<code>ls</code>	<code>Get-ChildItem</code>
<code>cat</code>	<code>Get-Content</code>
<code>history</code>	<code>Get-History</code>

Table 6-2. *Built-in PowerShell Aliases Emulating UNIX Commands (Continued)*

Name	Definition
pwd	Get-Location
ps	Get-Process
mv	Move-Item
mount	New-PSDrive
lp	Out-Printer
popd	Pop-Location
pushd	Push-Location
rm	Remove-Item
chdir	Set-Location
cd	Set-Location
sort	Sort-Object
sleep	Start-Sleep
kill	Stop-Process
tee	Tee-Object
echo	Write-Output

As you can see, the navigation commands are all there. Seasoned UNIX shell users will find that the habit of typing `ls` instead of `dir` will not break their commands. The `diff` command behaves a bit different than the UNIX text-line-oriented `diff` utilities in that it compares objects and their properties. It can be used on files too and provides similar features; the only caveat is that the output is not the standardized, unified `diff` format but PowerShell's difference objects converted to text. You can learn more about the `diff` command and other pipeline- and object-oriented command in Chapter 3.

I really like the way the `mount` command aliases the `New-PSDrive` cmdlet. Unfortunately, that is all it has in common with UNIX file system mounting. The syntax of adding a new drive and accessing it is completely different. To make matters even stranger, there is no built-in symmetric `umount` alias that will map to `Remove-PSDrive`. PowerShell drives are very different than drives that the operating system uses. To learn about how the shell uses drive providers to work with drives, please see to Chapter 7.

Note The PowerShell commands use quite a different foundation for their implementation than UNIX shell commands. For starters, all commands that traditionally use text use objects. File-oriented commands now work on items that can be something other than a file, say, a registry entry. Even though we have aliases with the same names defined for us, those aliases should be interpreted as learning tools: they are the trail that the PowerShell team have left for us that says, “Here, if you have used that command from a different shell, you should look into this PowerShell substitute.” The aliases are not a binding contract that guarantees compatibility; they just point us in the right direction.

Convenience Aliases

Apart from trying to emulate existing shells, PowerShell offers a host of new features. It also includes a good many aliases that help in using those new features. As a word of warning—these aliases may seem too cryptic for a new user and may feel as if they can obfuscate a command beyond recognition. However, many people find them useful, and many examples on the Net use them, so getting to know them may be worth the effort. The convenience aliases for built-in PowerShell commands are listed in Table 6-3.

Table 6-3. *Built-in PowerShell Convenience Aliases*

Name	Definition
ac	Add-Content
clc	Clear-Content
cli	Clear-Item
clp	Clear-ItemProperty
clv	Clear-Variable
cpp	Copy-ItemProperty
%	ForEach-Object
fc	Format-Custom
fl	Format-List
ft	Format-Table
fw	Format-Wide
gci	Get-ChildItem
gcm	Get-Command
gc	Get-Content
gi	Get-Item
gp	Get-ItemProperty
gl	Get-Location
gm	Get-Member
gps	Get-Process
gdr	Get-PSDrive
gsnp	Get-PSSnapin
gsv	Get-Service
gu	Get-Unique
gv	Get-Variable
gwmi	Get-WmiObject

Table 6-3. *Built-in PowerShell Convenience Aliases (Continued)*

Name	Definition
group	Group-Object
iex	Invoke-Expression
ii	Invoke-Item
mp	Move-ItemProperty
ni	New-Item
ndr	New-PSDrive
nv	New-Variable
oh	Out-Host
rp	Remove-ItemProperty
rdr	Remove-PSDrive
rsnp	Remove-PSSnapin
rv	Remove-Variable
rnv	Rename-ItemProperty
rvpa	Resolve-Path
select	Select-Object
sc	Set-Content
si	Set-Item
sp	Set-ItemProperty
sv	Set-Variable
sort	Sort-Object
sasv	Start-Service
spps	Stop-Process
spsv	Stop-Service
?	Where-Object
where	Where-Object

The most important of the aliases in Table 6-3 are the % alias, which maps to `ForEach-Object`, and `?`, which points to `Where-Object`. You might be thinking that using those aliases has the potential of truly bringing Perl's obfuscation capabilities to PowerShell. The fact is that those two are very easy to type on the console, and once your fingers learn them, you will not look back. Just remember not to use them in scripts and to shorten `ForEach-Object` and `Where-Object` to `foreach` and `where` in lengthier code snippets.

You can see that `select`, `where`, and `sort` alias `Select-Object`, `Where-Object`, and `Sort-Object` respectively, but there is no `foreach` alias that points to `ForEach-Object`. Yet that alias seems to be present. In reality, `foreach` coincides with the `foreach` loop statement in the PowerShell script language. This case is handled by the PowerShell's parser, which intelligently decides whether to interpret `foreach` as a loop statement or cmdlet invocation according to the context.

Summary

In this chapter, you learned how to define and use aliases to help you in your daily shell use. Command aliases are a powerful tool and a productivity booster. They require little investment and yield huge returns once you start using them. Use them with care, though, so that you do not get into a situation where you pull the rug from beneath yourself by modifying the core commands that are used by both the shell and all utilities. I showed you several of the dangerous scenarios and demonstrated techniques of getting out of those situations, just in case you do get into trouble.

After that, I introduced the built-in shell aliases and how to use them to leverage your existing shell scripting skills acquired using the DOS or UNIX shells. Those aliases are an important migration tool for users who have previous experience with text-based shells. They are what get us thinking, “Aha! This is what I should be using when I want to do that.”



Providers

Did you know that PowerShell has no concept of files or folders? That's why all the operations that involve files and folders seem to be referring to items: `Get-Item`, `Get-ChildItem`, `Get-ItemProperties`. When I first saw that `dir` was actually an alias for `Get-ChildItem`, I screamed. I felt as if the cmdlet naming convention has been conceived by a really sick mind. But, later as I was learning more about the shell, I found out that the reasoning behind the item concept is that PowerShell works with all types of objects that look like files. Instead of taking the UNIX shells' approach and calling everything a file, even things that are obviously not files, the language designers have come up with the item concept.

An item is an object that has content and properties and can contain other items. This definition easily accommodates files and folders: they are items with respective content and properties. This item concept is very powerful, because it can be applied to many objects on our system. Items are manipulated by providers—those are the actual objects that know how to create items, retrieve them, modify them, remove them, and so on. Providers are one of the official shell extensibility mechanisms. The shell comes with several providers built-in and a snap-in can install additional providers.

Enumerating Providers

Let's now take a virtual walk around Providerville. We can get a list of all providers by calling the `Get-PSProvider` cmdlet:

```
PS> Get-PSProvider
```

Name	Capabilities	Drives
----	-----	-----
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess	{C, D, E, F}
Function	ShouldProcess	{Function}
Registry	ShouldProcess	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}
Certificate	ShouldProcess	{cert}

As you can see, each of the providers has been used to mount a drive that looks like a regular file system drive. Of those drives, only the ones exposed by the `FileSystem` provider point to real files. Providers are automatically registered by the snap-ins that contain them, so

let's get the name of the snap-in and the .NET type that implements each of the providers. We will format the results in a list, because the `ImplementingType` property value can get long:

```
PS> Get-PSProvider | select Name,PSSnapIn,ImplementingType | fl

Name           : Alias
PSSnapIn       : Microsoft.PowerShell.Core
ImplementingType : Microsoft.PowerShell.Commands.AliasProvider

Name           : Environment
PSSnapIn       : Microsoft.PowerShell.Core
ImplementingType : Microsoft.PowerShell.Commands.EnvironmentProvider

Name           : FileSystem
PSSnapIn       : Microsoft.PowerShell.Core
ImplementingType : Microsoft.PowerShell.Commands.FileSystemProvider

Name           : Function
PSSnapIn       : Microsoft.PowerShell.Core
ImplementingType : Microsoft.PowerShell.Commands.FunctionProvider

Name           : Registry
PSSnapIn       : Microsoft.PowerShell.Core
ImplementingType : Microsoft.PowerShell.Commands.RegistryProvider

Name           : Variable
PSSnapIn       : Microsoft.PowerShell.Core
ImplementingType : Microsoft.PowerShell.Commands.VariableProvider

Name           : Certificate
PSSnapIn       : Microsoft.PowerShell.Security
ImplementingType : Microsoft.PowerShell.Commands.CertificateProvider
```

With the exception of the Certificate provider, each one of them is stored in the `Microsoft.PowerShell.Core` snap-in.

In a perfect world, we would have been able to work with registry keys in absolutely the same way as with files and vice versa. Unfortunately, the underlying storage type has its limitations, and that means that not all providers are created equal. Therefore, PowerShell designers have come up with the concept of provider capabilities, which allow a provider to declare to the outside world the operations it supports. You already saw the capabilities of the built-in providers in the previous example. Here are the possible capabilities a provider can support:

- **Credentials:** This is the ability to use credentials provided by the user on the command line to access an item.
- **ExpandWildCards:** If implemented, the provider should be able to expand paths that contain wildcard symbols. PowerShell already implements that functionality, and it will fall back to its implementation if the provider does not support the capability. Supporting the capability typically means better performance for a provider.

- **Include:** This is the ability to include items based on a wildcard string. Again, PowerShell knows how to handle wildcard strings when this capability is not available, possibly at the cost of performance.
- **Exclude:** Similar to Include, this is the ability to exclude items based on a wildcard. Providers implementing this capability will typically perform faster than those taking advantage of PowerShell's default implementation.
- **Filter:** This provides the ability to additionally filter items according to a provider-specific string.
- **None:** This explicitly specifies no additional capabilities. PowerShell will fall back on its implementation and capabilities inherited from the provider base classes.
- **ShouldProcess:** The provider calls the ShouldProcess method before performing modifications to the data store. This allows the user to use `-WhatIf` and `-Confirm` parameters and preview or manually confirm every data modification.

Drives

Providers would be useless if there were no way to get items from them. The standard mechanism to provide items is through a drive. Similar to Windows drives, PowerShell drives use an identifier followed by a colon like `C:` or `D:`. Unlike Windows drives, PowerShell drives can have longer identifiers like `Env:`, `cert:`, and `HKCU:`. We can get information about the drives currently registered on the system with the `Get-PSDrive` cmdlet:

```
PS> Get-PSDrive
```

Name	Provider	Root	CurrentLocation
----	-----	----	-----
Alias	Alias		
C	FileSystem	C:\	
cert	Certificate	\	
D	FileSystem	D:\	
E	FileSystem	E:\	
Env	Environment		
F	FileSystem	F:\	
Function	Function		
HKCU	Registry	HKEY_CURRENT_USER	Software
HKLM	Registry	HKEY_LOCAL_MACHINE	...indows
Variable	Variable		

Note that every drive has a root location and keeps track of its current location if it has been modified. Drives are not set in stone; we can create new drives and make it easier to access files using shorter paths. Here is how to add a `docs:` drive that points to the Documents folder:

```
PS> New-PSDrive -name docs -PSProvider FileSystem `
    -Root C:\Users\Hristo\Documents
```

Name	Provider	Root	CurrentLocation
docs	FileSystem	C:\Users\Hristo\Documents	

We can now navigate there and use the newly created drive:

```
PS> cd docs:
PS> dir
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\Hristo\Documents
```

Mode	LastWriteTime	Length	Name
d----	10/1/2007 10:55 PM		My Virtual Machines
d----	9/4/2007 10:46 PM		My Weblog Posts
d----	7/25/2007 10:39 AM		Virtual Machines
d----	10/2/2007 1:14 AM		Visual Studio 2005
d----	7/26/2007 7:04 PM		WindowsPowerShell

Note that drives are not completely transparent. Even though we are in the docs : drive, the directory path still points to the original location: C:\Users\Hristo\Documents. The FileSystem provider can work with network shares too. You map network shares as local drives, just as you would do it for a normal folder. Just provide the full network path to the share as the drive root. Here is how to map the \\srv1\public share as the local net : drive:

```
PS> New-PSdrive -Name net -PSprovider FileSystem -root \\srv1\public
```

Name	Provider	Root	CurrentLocation
net	FileSystem	\\srv1\public	

```
PS> cd net:
PS> dir
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::\\srv1\public
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	9/5/2007 10:37 PM		Code Snippets
d----	8/18/2007 4:32 PM		Downloads
d----	10/2/2007 12:05 AM		PowerShell

Removing mapped drives is a matter of calling `Remove-PSDrive`. Remember to navigate away from those drives, so that they are not in use; otherwise, the removal operation will fail:

```
PS> Remove-PSDrive net
Remove-PSDrive : Cannot remove drive 'net' because it is in use.
At line:1 char:15
+ Remove-PSDrive <<<< net
PS> c:
PS> Remove-PSDrive net
PS>
```

Drive Scope

Interestingly enough, drives obey scoping rules. Similar to rules for variables, a drive that is registered in a function or a script block will shadow a drive having the same name that has been declared before. Exiting the function or the script block will effectively unregister the drive. Here is a function that maps a drive and returns the root item:

```
PS> function Get-Documents()
{
    New-PSDrive -name docs -PSProvider FileSystem `
        -Root C:\Users\Hristo\Documents

    return (Get-Item docs:\)
}
```

```
PS> Get-Documents
```

Name	Provider	Root	CurrentLocation
----	-----	----	-----
docs	FileSystem	C:\Users\Hristo\Documents	

```
PSPath          : Microsoft.PowerShell.Core\FileSystem::C:\Users\Hristo\Documents\
...
```

```
PS> cd docs:
Set-Location : Cannot find drive. A drive with name 'docs' does not exist.
At line:1 char:3
+ cd <<<< docs:
PS>
```

We can use the `docs :` drive only from within the function. The error message we get when we try to access the drive from outside the function shows that the drive is not available anymore.

Navigating to Drives

You might have been wondering, from the examples you have seen so far, how we could use the `cd` command to switch to a new current drive. If you are an experienced `cmd.exe` user, you would not expect that to work—the `cd` command can only change to new directories on the current drive. Here is how that works when running `cmd.exe`:

```
C:\>cd d:\
```

```
C:\>
```

As you can see, the current drive remains set to `C:`. If you dig deeper, you will notice that `cmd.exe` keeps a “current” folder for every drive that it knows about. We can set the current folder to a drive other than the current one, and it will be available once we switch to that drive. For example, if we are currently in the `C:\` location and `cd` to `D:\tmp`, we will change the current folder for the `D:` drive but will not navigate to that location. We can see that we really changed the current folder for the `D:` drive only after switching to it.

```
C:\>cd D:\tmp
```

```
C:\>D:
```

```
D:\tmp>
```

There are no such things as drive-specific current folders in PowerShell. There is only one current location, and it applies to all drives. That means we can change drives using the `cd` command, which is really an alias for the `Set-Location` cmdlet. Here is how that works:

```
PS> Get-Location
```

```
Path
```

```
----
```

```
C:\
```

```
PS> Set-Location D:\tmp
```

```
PS> Get-Location
```

```
Path
```

```
----
```

```
D:\tmp
```

In fact, the `C:` and `D:` commands and so on are not built-in shell commands. They are function names (yes, the colon character is valid for function names!), and PowerShell, by default, defines 26 functions that allow us to switch to letter drives. Here is how to get them:


```
PS> Get-Command [A-Z]:
```

CommandType	Name	Definition
Function	A:	Set-Location A:
Function	B:	Set-Location B:
Function	C:	Set-Location C:
...		
Function	Z:	Set-Location Z:

You can see that the functions call Set-Location to navigate to the respective drive.

PROVIDER-QUALIFIED PATHS

Drives are the most convenient way to reference an item; they allow us to map an item and access it with shorter paths. Drives are not the only way to access items, though. Each provider exposes a full provider-qualified path that usually takes the form of [ProviderName]::[Provider-local path]. Let's take a look at how those paths look for the file system provider:

```
PS> dir C:\
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\
```

Mode	LastWriteTime	Length	Name
d----	9/5/2007 10:37 PM		<DIR> Code Snippets
d----	8/18/2007 4:32 PM		<DIR> Downloads
d----	6/29/2007 4:10 PM		<DIR> inetpub
. . .			

See the Microsoft.PowerShell.Core\FileSystem::C:\ header? This is the provider-qualified path that points to the C:\ item served by the FileSystem provider. Here is how that looks for the HKEY_CURRENT_USER registry hive:

```
PS> dir HKCU:\
```

```
Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
```

SKC	VC	Name	Property
0	1	.AUP	{{default}}
2	0	AppEvents	{}
. . .			

We get the `Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER` path, which we can use to navigate to that location. The best part is that that path works without having a drive mapped to that location:

```
PS> Remove-PSDrive HKCU
PS> cd Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
HKEY_CURRENT_USER
PS> Get-Location

Path
----
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
```

As you can see, the current folder looks and works like a normal drive-qualified path. Note that we removed the `HKCU:` drive in the example for testing purposes; the simplest way to get it back is to restart your shell session.

Provider Capabilities

I've already mentioned that not all providers are created equal. The built-in providers have different capabilities, and some providers may not support all item-related cmdlets. Custom providers are in the same situation. We will describe several categories of providers according to their capabilities and the cmdlet groups they support. Under the hood, the various providers, having different capabilities, are implemented as classes inheriting from base classes, which do or don't implement several interfaces. While listing the provider capability types, I will mention how the different built-in providers support those cmdlets and list any peculiarities.

Basic Provider Capabilities

The absolute minimum for a provider is to be able to properly initialize the shell session and register with it. This provides no useful functionality to cmdlets but lets the provider appear in the report returned by `Get-PSProvider`.

Drive Providers

Providers having the drive capability can register and unregister new drives; they support the `New-PSDrive` and `Remove-PSDrive` cmdlets. A provider that cannot register a drive is, of course, useless, and all built-in providers support those functions.

Item Providers

These are providers that can work with items. They support the following cmdlets:

- `Clear-Item` clears the content of the item and replaces it with an empty value. The empty value is provider specific: it can be an empty value when working with files, no values and subkeys when working with the registry, or `$null` when working with variables.
- `Get-Item` gets a reference to an item given a location.

- `Invoke-Item` invokes the default action for the item at the specified path. It usually treats the file as executable and runs or opens it with the associated application.
- `Set-Item` sets a new value for an item. Its implementation is provider specific: it can change a file's contents, set a new variable value, or set a value for the default value for a registry key.
- `Resolve-Path` expands wildcards and returns paths that match the wildcard expression.
- `Test-Path` verifies if a path exists and returns `$true`; otherwise, `$false`.

Being able to register drives and work with items is the minimum a provider has to support to be of any use. Of the built-in providers, every one supports these capabilities.

Item Container Providers

Item container providers can work with hierarchical items and can return items that contain other items. Here are the cmdlets that are supported by this provider type:

- `Copy-Item` copies an item from one parent location to another.
- `Get-ChildItem` gets all child items for the given location. This is how the `FileSystem` provider `dir` command is actually implemented. In fact, `dir` is an alias for `Get-ChildItem`. The `Environment`, `Variable`, `Function`, and `Alias` providers implement this functionality in a somewhat strange way. They will always return the item at the specified location when you ask for its children. `Get-ChildItem` behaves in exactly the same way `Get-Item` does for these four providers: they are not really hierarchical, but they are considered item container providers (because they use the other container functionality: copying, renaming, creating, and removing items). It is very likely that the funny `Get-ChildItem` implementation is there so that we do not get an error. Here is an example:

```
PS> cd alias:
```

```
PS> Get-ChildItem dir
```

CommandType	Name	Definition
-----	----	-----
Alias	dir	Get-ChildItem

```
PS> cd env:
```

```
PS> Get-ChildItem path
```

Name	Value
----	----
Path	C:\Windows\System32\WindowsPowerShell...

- `New-Item` will create a new item at the specified location and possibly set a value. The item can be re-created if it already exists if the `-Force` parameter is used.
- `Remove-Item` deletes an item at the specified location. The familiar file deletion operation `del` is actually an alias for `Remove-Item`.
- `Rename-Item` renames the item at the specified location.

All built-in providers are item container providers. Some of them do not really have a hierarchical structure, as shown in the `Get-ChildItem` example.

Navigation Providers

Navigation providers support advanced operations that involve more than one item container. They support the following cmdlets:

- `Join-Path` takes two path components and joins them in a single path. It can optionally resolve the newly created path and raise an error if it does not exist.
- `Move-Item` moves an item from its current container to the specified location.

Of the built-in providers, only `FileSystem`, `Registry`, and `Certificate` are navigation providers. Trying to use one of those operations for paths in drives registered by other providers will result in an error. Here is what happens when we try to move an alias:

```
PS> Move-Item alias:\dir alias:\dir2
Move-Item : Cannot retrieve the dynamic parameters for the cmdlet. The
NavigationCmdletProvider methods are not supported by this provider.
At line:1 char:10
+ Move-Item <<<< alias:\dir alias:\dir2
```

Item Content Providers

Item content providers support operations involving an item's contents, for example, changing, adding to, and clearing them. Here are the cmdlets that use the item content capabilities:

- `Add-Content` adds the value to the specified item's content.
- `Clear-Content` sets the item's content to an empty value. The empty value is provider specific.
- `Get-Content` retrieves the content of the specified item.
- `Set-Content` sets the content of the specified item, destroying all old content in the process.

Almost all built-in providers are item content providers. The exceptions are the `Registry` and `Certificate` providers.

Item Property Providers

Item property providers allow items to have and expose properties that can be retrieved and modified. The cmdlets that rely on these capabilities follow:

- `Clear-ItemProperty` deletes the value of a property and sets it to a provider-specific empty value. The property itself is not deleted.
- `Get-ItemProperty` retrieves the item property value.
- `Set-ItemProperty` sets a new value for the specified item property.

Only two providers support the item property capability: `FileSystem` and `Registry`.

Dynamic Item Property Providers

Providers supporting the dynamic item property capabilities can have properties added and removed dynamically at runtime. They also support copying and moving properties. Here are the cmdlets available for working with that capability:

- `Copy-ItemProperty` copies a property and its value from one item to another.
- `Move-ItemProperty` moves a property and its value from one item to another.
- `New-ItemProperty` creates a new property for the specified item. It can optionally set a value for the new property.
- `Remove-ItemProperty` removes a property and its value from the item.
- `Rename-ItemProperty` renames a property preserving its value.

The dynamic item property capability is a strange one. None of the built-in providers support it except the Registry provider. It seems that the capability has been created specifically for the Registry provider where properties are used to represent registry key values and values can be created, removed, copied, and moved.

Item Security Descriptor Providers

Some providers allow for an item to have a security descriptor or an access control list, so that it can be protected against unauthorized access. That item can be a file or some other operating system object. There are just two cmdlets that use this capability:

- `Get-Acl` gets the access control list (ACL) for the item at the specified location.
- `Set-Acl` sets a new ACL for the item at the specified location.

This capability is supported by only the `FileSystem` and `Registry` providers. That means that we can read and modify the access control lists for both files and registry keys.

Summary

Providers are a wonderful mechanism of abstracting all types of interaction with the outside world. In this chapter, you learned to register and use newly created drives to make paths shorter and save keystrokes. We then explored the built-in providers and their capabilities. Having an idea about what capabilities are supported in a provider helps you to know which cmdlets can be used on items that reside inside drives registered from that provider.

Taking the time to make yourself comfortable with using the default providers is definitely worth the effort. In Chapter 10, we will be using the `Certificate` provider to get information about registered certificates when we are signing our scripts. Chapter 21 discusses the PowerShell Community Extensions project, which ships several providers that make working with Active Directory, the Internet Explorer 7 RSS feed store, and the .NET global assembly cache a lot easier.



Script Files

I started learning about computers and programming by playing with small BASIC programs on an Apple II-compatible computer. After making my first steps in the language and the environment, I started typing up programs that I found in magazines or books. Most often, those were little games that I could type in an hour or so and, if I had not made a mistake while doing that, I could play the game for some time afterward. I typed BASIC code without a text editor or anything like that; I entered raw code at the console. My biggest problem in doing that was that I did not know how to save the program to a floppy disk, so that I could run it later. I had to learn how to do that in order to build larger programs.

PowerShell's script language offers much more power and is much more expressive than that long-forgotten BASIC dialect. Hey, it was built close to thirty years later—it has to be better! Even armed with that power, we still need a way to save our code and reuse it later. Imagine trying to write a twisted command that has a long chain of parameters. You wrestle with it, go through the documentation, try several approaches, and finally, you nail it. You do not need to remember every parameter you just learned by heart—all you need is to save the command to a script file and run that when you need that command again.

We need script files when we want to create small script utilities that we can run on a regular basis. We also need to use them in those cases when we end up creating complex modular scripts that keep their code in separate files to make developing and maintaining everything manageable. In this chapter, you will learn how to handle all of those scenarios. We will go through the process of creating script files, passing parameters, returning values, and building modular scripts that span several files.

Creating Scripts

The typical PowerShell script is a text file that we can create using various tools. By default, those files carry the `.ps1` file extension. You can create them using Notepad, but it is best if you use a more powerful tool, one of the programmer text editors that offer features like syntax highlighting and even IntelliSense-like word completions. My favorite editor, which helped me with the examples for this book, is Notepad++, a free `notepad.exe` replacement that knows about many programming languages and offers syntax highlighting, word completions, and much more. You can even create scripts from the PowerShell console using here strings and the `Set-Content` cmdlet. Here is how to create our very first script:

```
PS> $code = @"
Write-Host "Hello, world!"
"@

PS> Set-Content hello-world.ps1 $code
PS> type hello-world.ps1
Write-Host "Hello, world!"
```

As you can see the `hello-world.ps1` file is a plain text file, and we can inspect its contents using the `type` command (which is really an alias of `Get-Content`).

Invoking Scripts

Now that we have our first script in place, we can go ahead and invoke it. First, let's see how PowerShell looks for files to invoke: all it does is look in the `PATH` environment variable, and interestingly enough, the current folder is not in the system `PATH`. That means that invoking a script in the current folder will require you to prefix it with the path, so the command in our case becomes `.\hello-world.ps1`. That will look familiar if you come from a UNIX background where a shell would not include the current folder in its `PATH` variable.

`cmd.exe` takes a different approach and includes the current folder in the `PATH`. If you are a seasoned `cmd.exe` user, that peculiarity may require some time getting used to. The rationale behind this behavior is that it is a security risk to have the current folder in the `PATH`. If the inverse were the case, programs in the current folder could shadow default programs and may allow an attacker to execute malicious code. Suppose that a malicious user, who can write to a shared folder, creates a script or executable with the same name as a legitimate utility. If the administrator visits that folder, logged in with his or her administrator credentials, and tries to invoke that utility, he or she would execute the malicious program in the current folder, allowing the attacker to run code under administrator credentials, thus elevating his or her privileges. That is why both PowerShell and UNIX shells require you to explicitly state that you want to execute a program from the current folder by prefixing the program name with the path specifier, `.\`.

Now that we are talking about security, you may have tried to invoke our `hello-world.ps1` script file and may have gotten the following error:

```
PS> .\hello-world.ps1
File D:\deshev\Writing\PowerShell book\Text\08 - Script Files\hello-wo
rld.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:17
+ .\hello-world.ps1 <<<<
```

This is PowerShell's secure-by-default building principle in action. The shell has several execution policies that configure the security level and user privileges for running scripts. By default, the shell runs in the `Restricted` policy level, which means that no scripts are allowed to execute. We can check the execution policy level by calling the `Get-ExecutionPolicy` cmdlet:

```
PS> Get-ExecutionPolicy
Restricted
```


We need to change that to run scripts. The preconfigured script execution policies for PowerShell are the following:

- **Restricted:** No script will be run under any circumstances. No matter if you are running under administrator user credentials or not, PowerShell will refuse to run your scripts. Only interactive commands, typed at the console, will be executed.
- **AllSigned:** PowerShell will run only scripts that have been signed by a trusted third party. To get details about how script signing works and how to sign a script, please see Chapter 10.
- **RemoteSigned:** The shell will require a signature from a trusted third party for all scripts that come from the Internet, whether downloaded from sites or received by e-mail. Locally created scripts will run without needing to be signed.
- **Unrestricted:** All scripts are run without any restrictions with regard to origin or a valid signature.

The **RemoteSigned** execution policy is a good combination of both a good level of security and ease of use. We will switch to that level by calling the `Set-ExecutionPolicy` cmdlet. Note that this cmdlet has to be run by an administrator user. Windows Vista users will need elevated privileges too. The easiest way to ensure elevated privileges is to right-click the PowerShell shortcut icon and choose “Run as administrator” from the context menu. Now that you have a shell session running with elevated privileges, you can run this command to change the execution policy:

```
PS C:\> Set-ExecutionPolicy RemoteSigned
PS C:\> Get-ExecutionPolicy
RemoteSigned
```

We have successfully switched to a less restrictive policy level, and it is now time to run our script. You do not need elevated privileges anymore, so it is best that you start a normal shell session now. Our script is a local one, so it does not need to be signed, and here it is

```
PS> .\hello-world.ps1
Hello, world!
```

You can omit the `.ps1` extension when calling a script and provide the file base name only, like this:

```
PS> .\hello-world
Hello, world!
```

In addition, the familiar script block execution operator, `&`, works for script files too:

```
PS> & .\hello-world.ps1
Hello, world!
PS> & .\hello-world
Hello, world!
```

Passing Parameters

To paraphrase a popular saying, “No script is an island.” Scripts do not exist by themselves; they need to interact with the outside world. They can accept parameters as a way of specifying actions that need to be performed, action targets, and whatnot. Similar to functions and script blocks, scripts have the `$args` variable set up with the parameters passed at the time of their invocation. Let’s use that to create a small script that looks for music files matching a specified wildcard inside a given folder:

```
$where = $args[0]
$what = $args[1]

if (!$what -match "\.mp3$")
{
    $what = $what + ".mp3"
}
Get-ChildItem $where $what -Recurse
```

The script augments the `$what` parameter by adding the `.mp3` extension if the wildcard does not already end with it. That will limit the search to only files with the `.mp3` extension. The code will recursively enumerate all files and return the ones matching the `$what` wildcard. We save the preceding code to a file named `Get-Music.ps1`. Here is how I use it to rummage through my “The Cranberries” music collection:

```
PS> .\Get-Music.ps1 'D:\Music\The Cranberries' *salvation*
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::D:\Music\The Cranberries\1996 - To The Faithful Departed
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	2/5/2006 1:36 PM	2373632	02 Salvation.mp3

```
Directory: Microsoft.PowerShell.Core\FileSystem::D:\Music\The Cranberries\Bonus CD - Live in Stockholm - Limited edition
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	2/5/2006 1:37 PM	2449536	04-Salvation (live).mp3

Of course, just like functions and script blocks do, script files support the `param` statement that allows us to name our parameters and avoid doing manual value extraction from the `$args` collection. Here is how we can rewrite our `Get-Music.ps1` script using named parameters:

```
param ($where, $what)

if (!$what -match "\.mp3$")
{
    $what = $what + ".mp3"
}
Get-ChildItem $where $what -Recurse
```

Now, we save the preceding code snippet to the `Get-Music-Params.ps1` file. The `param` technique allows us to name our parameters at the script invocation site too:

```
PS> .\Get-Music-Params.ps1 -where 'D:\Music\The Cranberries' `
    -what *zombie*
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::D:\Music\The Cranberries\1994 - No Need To Argue
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	2/5/2006 1:36 PM	5865472	04 Zombie.mp3

Processing pipeline input is done by defining `begin`, `process`, and `end` blocks, again just like script blocks and functions. Here is a small script that gets a bunch of files and filters out any that are smaller than the given size:

```
param ($sizeLimit)

begin
{
}

process
{
    if ($_.Length -ge $sizeLimit)
    {
        $_
    }
}

end
{
}
```

Save the code to a file called `Filter-Smaller.ps1`. The script accepts a size limit variable that is then used to test every item in the pipeline and include the item in the result if the file size (the `Length` property) is greater than or equal to the size limit. All the action happens in the

process block, as it is invoked for every item in the input pipeline. The begin and end blocks do nothing in this example, but I like to include them, even if empty, for completeness. Here is how to get the scripts larger than 130 bytes using the preceding script:

```
PS> dir *.ps1 | .\Filter-Smaller.ps1 130
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::D:\deshev\Writing
\PowerShell book\Text\08 - Script Files
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	10/13/2007 1:02 AM	137	Get-Music.ps1

While we are discussing parameters, we should mention variables too. Scripts have access to all variables that have been defined by the shell user or by other scripts. Much like functions, scripts create a separate variable scope—and that means that they can read variables from their invocation site and its parent scopes, but writing to the same variables will shadow the parent ones. Let's see how that new variable scope may affect our code. First, let's create a script that accesses variables from its parent:

```
PS> $name = "John"
PS> Set-Content Variable-Test.ps1 "Write-Host $name"
PS> .\Variable-Test.ps1
John
```

What happens when we try to modify the variable's contents? A new variable gets created that shadows the original one. This way, the original value is protected from accidental overwrites by scripts that we call. Let's see that in action too—first, the script file, `Variable-Test.ps1`, which modifies the `$name` variable:

```
Write-Host "Within script original: $name"
$name = "Mike"
Write-Host "Within script modified: $name"
```

Next, we can call it:

```
PS> $name = "John"
PS> .\Variable-Test.ps1
Within script original: John
Within script modified: Mike
PS> Write-Host "Outside script: $name"
Outside script: John
```

As you see, the variable appears to be modified for the inner script only. We can modify variables in the parent scope, but we have to explicitly request that by using the `$global` scope prefix or the `Get-Variable/Set-Variable` cmdlet. Let's look at the first approach in the following `Variable-Test2.ps1` script code:

```
Write-Host "Within script original: $global:name"
$global:name = "Mike"
Write-Host "Within script modified: $global:name"
```

Calling it gets us different results than the previous script:

```
PS> $name = "John"
PS> .\Variable-Test2.ps1
Within script original: John
Within script modified: Mike
PS> Write-Host "Outside script: $name"
Outside script: Mike
```

The Set-Variable approach has to use the `-Scope` parameter to modify variables from the parent scope. That works in the same way as with functions and script blocks, and you can refer to Chapters 4 and 5 for examples.

Often, we want to isolate a variable or function so that it is global to a specific script. We may want all functions and script blocks defined in the script file to be able to access that variable or function, but we may explicitly want to make that object inaccessible outside the script. We cannot use the `$global` scope prefix, as that will expose a variable or function to all code no matter if it is defined in our script file. The solution is to use the `$script` scope prefix. To demonstrate this, we use our `Variable-Test3.ps1` script:

```
$script:name = "Jeremiah"

function Modify-Name()
{
    $script:name = "Mike"
    Write-Host "Within script function scope: $script:name"
}

Write-Host "Original script global scope: $script:name"
Modify-Name
Write-Host "Modified script global scope: $script:name"
```

Our script has two scopes: the script global scope and the `Modify-Name` function scope. No matter where we are, we can reference the script global name variable by referring to it as `$script:name`. Executing the script does not expose the script global variable to code outside the script:

```
PS> $name = "John"
PS> .\Variable-Test3.ps1
Original script global scope: Jeremiah
Within script function scope: Mike
Modified script global scope: Mike
PS> Write-Host "Outside script: $name"
Outside script: John
```

As you can see, the `$name` variable remains unaffected.

We also often want a variable or function that a script defines to be accessible in the current scope; in other words, we want to allow a script access to all our current variables and functions. We do that by prefixing the script name with a dot (.)—this is known as dot-sourcing a script. Recall our original Variable-Test.ps1 example script:

```
Write-Host "Within script original: $name"
$name = "Mike"
Write-Host "Within script modified: $name"
```

If we dot-source the script, it will modify the current \$name variable instead of creating another one:

```
PS> $name = "John"
PS> . .\Variable-Test.ps1
Within script original: John
Within script modified: Mike
PS> Write-Host "Outside script: $name"
Outside script: Mike
```

Note that it is important to leave a space between the dot and the script name. Dot-sourcing is a useful technique that we will employ later in this chapter to move function declarations to outside scripts and make them available to the current code.

Returning Values

Parameters are, most of the time, a one-way communications mechanism that will only pass data from the environment to the script. We need to be able to return values from our scripts. Sure, we can take the results of our computation and slap them in a global variable that can later be accessed from the outside world, but this would be like denying that the last forty years or so of computation research really existed. PowerShell provides a good way to return and output values from a script, and that is what we should use.

An object that is not explicitly consumed by code either by being bound to a variable or piped or redirected to another command is output to the pipeline for the next command. We can use that to generate a bunch of objects from our scripts and output them. Here is a script that outputs three temporary file names:

```
PS> $code = @"
File1.tmp"
File2.tmp"
File3.tmp"
"@

PS> Set-Content Generate-TempFiles.ps1 $code
PS> $files = .\Generate-TempFiles.ps1
PS> $files
File1.tmp
File2.tmp
File3.tmp
```

```
PS> .\Generate-TempFiles.ps1 | foreach { "File: " + $_ }
File: File1.tmp
File: File2.tmp
File: File3.tmp
```

As you can see, the values can be both assigned to a variable or can be passed down the pipeline to a command like `foreach`.

A `return` statement outputs an object and stops execution. Let's see how an explicit `return` differs from a simple output operation:

```
PS> $code = @"
return "File1.tmp"
return "File2.tmp"
"@

PS> Set-Content Generate-TempFiles2.ps1 $code
PS> .\Generate-TempFiles2.ps1
File1.tmp
```

Execution stopped after the first `return` statement, and we got the first object only.

The `return` statement will exit the script if it is executed at the top-level scope. In fact, a `return` operation will just exit the current scope, which means if we execute a `return` from a script block or a function inside our script, we will exit only that script block or function. The script will continue executing. To explicitly stop the script execution, we have to use the `exit` statement. Here is how we can exit a script from within a function. Our code is placed in the `Explicit-Exit.ps1` file that looks like this:

```
function Verify-TextFile($file)
{
    if (!$file.EndsWith(".txt"))
    {
        Write-Host "$file is not a text file"
        exit
    }
}

$files = "Names.txt", "Names.doc", "Names2.xls"
foreach ($file in $files)
{
    Verify-TextFile $file
}
```

Running the script produces the following output:

```
PS> .\Explicit-Exit.ps1
Names.doc is not a text file
```

As you can see, we never got to the last item, `Names2.xls`—script execution ended when we processed `Names.doc`.

Executing PowerShell Scripts from Other Environments

No matter how strongly we wish it, the entire world is not using PowerShell—yet! We will have to integrate our PowerShell code with existing solutions. We can write a useful script and execute it from a Windows scheduled task. We can transition an existing scripting solution piece by piece by developing new parts in PowerShell or porting existing functionality to PowerShell and integrating them with the existing solution by using command execution. All automation environments, task scheduler services, software build systems, and so on can execute an external command. That is where we can plug in a PowerShell script.

How do we run a PowerShell script from an external program then? Let's try to do it from within `cmd.exe`. I, and I suspect many experienced users, would expect that it would be sufficient to invoke `powershell.exe` and pass the name of the script as a parameter—that's not so:

```
H:\08 - Script Files>powershell.exe hello-world.ps1
The term 'hello-world.ps1' is not recognized as a cmdlet, function, operabl
e program, or script file. Verify the term and try again.
At line:1 char:15
+ hello-world.ps1 <<<<
```

Again, we are bitten by the fact that the current folder is not in the system `PATH` for PowerShell. We can fix things by using the `.` prefix to tell PowerShell it has to look in the current folder:

```
H:\08 - Script Files>powershell.exe .\hello-world.ps1
Hello, world!
```

Things get a bit complex when you get to read PowerShell's documentation and realize that `powershell.exe` actually does not support getting script names at the command line. What it really does is accumulate all its parameters and treat them as script code. This is best illustrated by a script file whose name contains spaces or other characters that PowerShell will interpret differently. Here is what happens:

```
H:\08 - Script Files>powershell.exe ".\hello world.ps1"
The term '.\hello' is not recognized as a cmdlet, function, operable progra
m, or script file. Verify the term and try again.
At line:1 char:8
+ .\hello <<<< world.ps1
```

PowerShell treats `.\hello` as a different command and complains that it cannot find it. We have to use some trickery to get around this: we have to wrap the script name in quotes and use the execute operator, `&`, to run the script:

```
H:\08 - Script Files>powershell.exe "& '.\hello world.ps1'"
Hello, world!
```

A lot of users on the Internet, me included, think this is ugly as hell. Hopefully, the PowerShell folks at Microsoft have recognized that as a problem and will provide a better way to call scripts in the next version of the shell.

Now that we know how to run PowerShell scripts from other programs, we are left with providing parameters and collecting output just as we would do with any other console program.

PowerShell allows us to create well behaved console programs that can return an exit code and use that to signal that something special happened to the outside world. We can do that by passing an exit code to the exit statement—that will both exit the currently running script and pass the error code to the environment. Here is how to write a script that returns 1 if it is given a text file, 0 otherwise. We will call the script `Is-TextFile.ps1`, and the code follows:

```
param ($file)

if ($file -like "*.txt")
{
    exit 1
}
else
{
    exit 0
}
```

Running the script from within `cmd.exe` and inspecting the `ERRORLEVEL` global variable that contains the exit code yields the following:

```
H:\08 - Script Files>powershell.exe "& '.\Is-TextFile.ps1' test.txt"
```

```
H:\08 - Script Files>echo %ERRORLEVEL%
1
```

```
H:\08 - Script Files>powershell.exe "& '.\Is-TextFile.ps1' Is-TextFile.ps1"
```

```
H:\08 - Script Files>echo %ERRORLEVEL%
0
```

We can use this technique to signal errors and special cases to programs using our scripts. This is the de facto standard mechanism to signal an error from a console application, and many environments support verifying at the process exit code.

Developing and Maintaining Script Libraries

Sooner or later, you will be tasked with creating a bigger and more complex solution that uses PowerShell as its implementation language. Before you know it, you can get in a situation where every piece of code is jammed in a single file and you can no longer find your way around in the code that you wrote yourself. This is an unpleasant situation to be in, and you have to be aware of your options for separating script code across several files to make things more manageable. In addition to being able to quickly navigate to the right spot in the right file, splitting code in several pieces makes it more reusable. It is quite common for a useful function to grow out of its first client code, get moved to another file, and later be included in many files that use it. In practice, I usually end up with a folder in my system `PATH` that contains useful scripts and scripts with useful functions that I can easily include in any script to start producing code that works in no time.

Dot-sourcing as a Means to Include Script Libraries

Script libraries are normal files that contain useful functions. They are ordinary PowerShell scripts that do not execute an action—they just define several functions and let the library client code call those to do the real job. Here is a sample library script, `Library1.ps1`:

```
function LibHello()
{
    Write-Host "LibHello2"
}

Write-Host "Library1.ps1 script included"
```

Including or importing a script library in a script file is a matter of executing it. That is why good programming practice insists that script libraries containing no executable code—function definitions only. Our sample library contains a small diagnostic message that we should delete from a real library.

Here is how we import the library from a script, `Lib-User-DotSource.ps1`, and call its `LibHello` function:

```
. .\Library1.ps1

LibHello
```

As you can see, we do nothing other than dot-sourcing the `Library.ps1` script. Keep in mind that the space between the initial dot and the script name is important; you will get an error if you omit it. Here is the output:

```
PS> .\Lib-User-DotSource.ps1
Library1.ps1 script included
LibHello2
```

We first execute the library script and get the diagnostic message and then execute the function code. We can use this technique to split our code and test if it works in isolation. For example, we can include a single library file in a different test script and test if the functions defined in the library work correctly.

The Library Path Problem

The preceding technique has one severe disadvantage: it treats all paths to library scripts as relative paths with respect to the current folder. It is not a problem of the technique per se; this is how PowerShell works. Assigning blame aside, this is a big problem because the script will break when you execute it from a different folder. Here is what happens if we execute the previous script from the upper directory:

```
PS> cd ..
PS> & 'Z:\08 - Script Files\Lib-User-DotSource.ps1'
The term '.\Library1.ps1' is not recognized as a cmdlet, function, operable program, or script file. Verify the term and try again.
```

```
At Z:\08 - Script Files\Lib-User-DotSource.ps1:1 char:2
+ . <<<< .\Library1.ps1
The term 'LibHello' is not recognized as a cmdlet, function, operable
program, or script file. Verify the term and try again.
At Z:\08 - Script Files\Lib-User-DotSource.ps1:3 char:9
+ LibHello <<<<
```

Chaos ensues, because the shell looks for `Library.ps1` in the current folder and does not find it, which causes a nonterminating error. Execution continues with the assumption that the `LibHello` function does not exist, which generates another error.

To solve this problem, we need a way to base our library inclusion paths to the folder of the current script to predictably include library files and not have that break when the script is called from another directory. How do we do that? PowerShell has a built-in variable called `$MyInvocation` that can help us here. This variable contains information about the current command invocation and is set whenever we invoke a script, function, or a script block. Let's create a script, `ScriptDirectory-Test.ps1`, that will return the current invocation details from within the script. Here is the script code:

```
return $MyInvocation
```

Now, pass that object to the `Get-Member` cmdlet to see what we are really getting back:

```
PS> .\ScriptDirectory-Test.ps1 | Get-Member -MemberType Property
```

```
TypeName: System.Management.Automation.InvocationInfo
```

Name	MemberType	Definition
InvocationName	Property	System.String InvocationName {get;}
Line	Property	System.String Line {get;}
MyCommand	Property	System.Management.Automation.CommandInf...
OffsetInLine	Property	System.Int32 OffsetInLine {get;}
PipelineLength	Property	System.Int32 PipelineLength {get;}
PipelinePosition	Property	System.Int32 PipelinePosition {get;}
PositionMessage	Property	System.String PositionMessage {get;}
ScriptLineNumber	Property	System.Int32 ScriptLineNumber {get;}
ScriptName	Property	System.String ScriptName {get;}

We get an `InvocationInfo` object, and we are interested in the `MyCommand` property that holds a `CommandInfo` object. Let's pass it to `Get-Member` instead to see what properties it holds for us by modifying our test command:

```
PS> (.\ScriptDirectory-Test.ps1).MyCommand | `
Get-Member -MemberType Property
```

```
TypeName: System.Management.Automation.ExternalScriptInfo
```

Name	MemberType	Definition
-----	-----	-----
CommandType	Property	System.Management.Automation.CommandTypes Co...
Definition	Property	System.String Definition {get;}
Name	Property	System.String Name {get;}
Path	Property	System.String Path {get;}

Bingo! We get an `ExternalScriptInfo` that is a type of `CommandInfo` object. See the `Path` property? That is what we really need here. Let's now use it to get the proper script path:

```
PS> (.\ScriptDirectory-Test.ps1).MyCommand.Path
Z:\08 - Script Files\ScriptDirectory-Test.ps1
```

Now, we wrap this into a function that will get the script path and return only the directory name, which we will save that to a new script file: `Get-ScriptDirectory.ps1`. Here is the code:

```
function Get-ScriptDirectory
{
    $invocation = $script:MyInvocation
    Split-Path $invocation.MyCommand.Path
}
```

```
Write-Host (Get-ScriptDirectory)
```

And here is the result that we get when we execute it:

```
PS> .\Get-ScriptDirectory.ps1
Z:\08 - Script Files
```

The code needs some explaining here. Remember I told you that `$MyInvocation` changes every time we execute a command, be it a function, a script, or a script block? That is what happens when we call the `Get-ScriptDirectory` function. We do not need the function invocation details; we need the script invocation ones instead. We could have used `Get-Variable` (see Chapter 4 for details) to get the `$MyInvocation` from the parent scope, but we are not guaranteed that the function will be called from the script global scope, and the parent scope can be that of another function. That is why we use the `$script` scope prefix to get the script invocation details. Finally, we pass the path to the script through the `Split-Path` cmdlet that will return the directory part only.

With the `Get-ScriptDirectory` function in our toolbox, we can use it to include library scripts in a way that does not depend on the current folder. We will create a script, `Lib-User.ps1`, that uses the function to include the `Library.ps1` script we dot-sourced earlier in this chapter. Here is the `Lib-User.ps1` code:

```
function Get-ScriptDirectory
{
    $invocation = $script:MyInvocation
    Split-Path $invocation.MyCommand.Path
}
```

```
$dir = Get-ScriptDirectory
$path = Join-Path $dir "Library1.ps1"
. $path
```

```
LibHello
```

Running it is possible from any folder now:

```
PS> .\Lib-User.ps1
LibHello script included
LibHello2
PS> cd ..
PS> & '.\08 - Script Files\Lib-User.ps1'
LibHello script included
LibHello2
```

The Join-Path invocation and the dot-source operation after it still feel too verbose. To make the library script inclusion shorter, we can create another function, `Get-LibraryPath`, as that will allow us to use more concise syntax for including script libraries. This is how that function looks and how we can use it to rewrite the `Lib-User.ps1` script:

```
function Get-ScriptDirectory
{
    $invocation = $script:MyInvocation
    Split-Path $invocation.MyCommand.Path
}

function Get-LibraryPath($relativePath)
{
    $dir = Get-ScriptDirectory
    return Join-Path $dir $relativePath
}

. (Get-LibraryPath "Library1.ps1")

LibHello
```

Having those two functions really makes including a path-independent script library a matter of dot-sourcing the result of `Get-LibraryPath`: `. (Get-LibraryPath "Library1.ps1")`. This syntax is a lot more awkward than a simple dot-source operation, but the fact that it works all the time justifies the price of using an extra set of parentheses around a function call. It is best if you store those two function definitions in your shell profile script so that they are available all the time, and so you do not have to redefine them in all your scripts. For details on how to do that, see Chapter 11.

Summary

In this chapter, I started with the basics of creating and invoking scripts. Real scripters are never content with the basics, so I continued by demonstrating how to pass parameters, pipe objects, and return values from a script. We did that by comparing script files to functions and script blocks—when it comes to rules about passing parameters, processing pipeline input, and returning values, those three objects behave in the same way.

An important section of this chapter was dedicated to using PowerShell from other environments like `cmd.exe` and integrating it: you learned how to call a PowerShell script and how to return a meaningful exit code. This makes our script files good citizens in the console application world.

Last, but not the least, we went through the process of creating maintainable script libraries. We will use what we learned here in the next chapters, where we will be creating larger scripts that solve complex real-world problems.



Error Handling and Debugging

The first steps in any programming environment usually involve doing the classic “Hello, world” example, first introduced some time in the 1970s and now a time-honored tradition. This chapter might well start with a “Hello, cruel world” example, as it will deal with a lot of pain. Here is the place where we take the red pill and enter the real world of programs crashing and misbehaving. To make things even more exciting, let’s assume we are placed right in the middle of a production environment and everyone expects us to get things working.

No script or program is guaranteed problem-free execution. Anything can go wrong, and we have to be ready for that. In this chapter, you will learn how to handle errors raised by commands you issue and scripts that you execute. We will detect, react to, and suppress errors depending on our goal at the time. We will go to the other side too: I will show you how to raise errors if you detect an abnormal condition, bad input data, or something completely weird.

The second part of this chapter will focus on debugging our code. Programs and scripts tend to grow over time to provide more and more to their demanding users. What starts as a one-liner can easily grow until it no longer fits inside the skull of a single person—that is where errors or bugs start creeping in. Finding and killing bugs is an important skill that every person working with computers develops in time. Here, I will cover the tools that PowerShell offers that can help when trouble strikes. We will be working with diagnostics and error prevention techniques that can bring peace, discipline, and sanity to our script development routine.

Handling Errors

Errors in PowerShell can be of two kinds: terminating and nonterminating. The first kind of is similar to the exception concept found in many programming languages. Those are errors that are triggered because the command interpreter has encountered an exceptional situation and cannot continue. Once we raise such an error, all further execution stops. Typical terminating errors are raised by script syntax errors, trying to call nonexistent commands, or something of a similar severity. User-raised errors resulting from the `throw` statement, which we will look into later in this chapter, are also terminating errors.

Nonterminating errors are less severe. Those are usually errors indicating that the current command cannot be completed, but the issues that caused them are not important enough to stop further execution. PowerShell has a pipeline similar to the normal object output pipeline reserved only for errors. Nonterminating errors are written to that pipeline, and the default action there is to display them to the user using red colored text. A typical nonterminating error is a file delete operation: it may fail for various reasons, but most of the time, we would want to continue execution anyway.

Let's see that file delete error in action. We will define a function called `Raise-NonTerminatingError` that will trigger a nonterminating error from a function stored in the `Error-Termination.ps1` script file. Here is the script:

```
function Raise-NonTerminatingError
{
    del nosuchfile.txt
}
```

```
Write-Host "script start"
```

```
Raise-NonTerminatingError
```

```
Write-Host "script end"
```

It prints a diagnostic message before and after calling the `Raise-NonTerminatingError` function. Here is the result that we get after running the script:

```
PS> .\Error-Termination.ps1
script start
Remove-Item : Cannot find path 'Z:\09 - Error Handling and Debugging\nosuchfile.txt' because it does not exist.
At Z:\09 - Error Handling and Debugging\Error-Termination.ps1:3 char:8
+     del <<<< nosuchfile.txt
script end
```

Even though we get an error message, we still see our “script start” and “script end” messages printed to the console—that means script execution continues after the error. Let's now trigger a terminating error, from another function `Raise-TerminatingError` that we'll call from our script:

```
function Raise-NonTerminatingError
{
    del nosuchfile.txt
}
```

```
function Raise-TerminatingError
{
    del nosuchfile-terminating.txt -ErrorAction Stop
}
```

```
Write-Host "script start"
```

```
Raise-NonTerminatingError
Raise-TerminatingError
```

```
Write-Host "script end"
```


As you can see, we instruct the `del` command to stop all execution—that is, raise a terminating error if something goes wrong. Here is the result that we see after running our script again:

```
PS> .\Error-Termination.ps1
script start
Remove-Item : Cannot find path 'Z:\09 - Error Handling and Debugging\nosuchfile.txt' because it does not exist.
At Z:\09 - Error Handling and Debugging\Error-Termination.ps1:3 char:8
+     del <<<< nosuchfile.txt
Remove-Item : Command execution stopped because the shell variable "ErrorActionPreference" is set to Stop: Cannot find path 'Z:\09 - Error Handling and Debugging\nosuchfile-terminating.txt' because it does not exist.
At Z:\09 - Error Handling and Debugging\Error-Termination.ps1:8 char:8
+     del <<<< nosuchfile-terminating.txt -ErrorAction Stop
```

We get two errors this time. The nonterminating one that tries to delete a nonexistent file comes first. The second error is what makes the difference here: we do not see our “script end” output anymore. That means script execution stopped once the critical error was raised.

Common Error-Handling Parameters

All PowerShell cmdlets support a set of parameters related to error handling. We can run any cmdlet and instruct it to generate additional output or perform a specific action in case an error occurs. The parameters: `ErrorAction`, `ErrorVariable`, `Debug`, and `Verbose` are all a part of the common parameters that every cmdlet must support.

We have already seen the `ErrorAction` parameter in the preceding example. The parameter is an enumeration that tells the cmdlet what to do when an error occurs. The possible values follow:

- **Continue:** This is the default. The error is reported to the user and execution continues. We saw this parameter in action in the `Raise-NonTerminatingError` function.
- **Stop:** The error is reported to the user and execution stops. As we saw in the `Raise-TerminatingError`, this option causes the cmdlet to raise a terminating error that usually stops all further execution. The “usually” part here means that we can still set up an error trap that can catch and handle this error without terminating the script (you will see how to do that later in this section).
- **SilentlyContinue:** Errors are ignored. This option is similar to `Continue` with the only difference that it does not write the error to the user. This is the perfect option for suppressing errors that have no real significance and hiding them from the user. Here is how to ignore a file deletion failure:

```
PS> del nosuchfile.txt -ErrorAction SilentlyContinue
PS>
```

- **Inquire:** This option causes the shell to prompt the user and let him or her decide what to do next. Here is how our, now favorite, file delete operation looks when we set this option:

```
PS> del nosuchfile.txt -ErrorAction Inquire
```

```
Confirm
```

```
Cannot find path 'Z:\09 - Error Handling and Debugging\nosuchfile.txt'
because it does not exist.
```

```
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help
(default is "Y"):
```

Answering Yes or Yes to All is the same as choosing the Continue action: you will get an error message, but execution will continue. The Halt Command option behaves like the Stop error action: you get a terminating error. The Suspend option is interesting: execution is temporarily suspended and you are thrown a fully functional nested prompt that you can use to maybe perform actions you forgot before running the command or look around trying to find out what caused the error. The Inquire option is a powerful debugging tool that you will learn how to use later on in this chapter. Type exit to return to the command and the same prompt:

```
PS> del nosuchfile.txt -ErrorAction Inquire
```

```
Confirm
```

```
Cannot find path 'Z:\09 - Error Handling and Debugging\nosuchfile.txt'
because it does not exist.
```

```
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help
(default is "Y"):S
```

```
PS> pwd
```

```
Path
```

```
----
```

```
Z:\09 - Error Handling and Debugging
```

```
PS> exit
```

```
Confirm
```

```
Cannot find path 'Z:\09 - Error Handling and Debugging\nosuchfile.txt'
because it does not exist.
```

```
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help
(default is "Y"):
```

Choosing whether to terminate our script is one thing, but getting hold of the real error is a completely different matter. To get the error object that is generated by a cmdlet, we can provide the `ErrorVariable` parameter. Most often, we need the error object when we use an `ErrorAction` of `SilentlyContinue` so that we hide errors from our user but still know if an error occurred. We can achieve that by checking if the variable that we configured as an error container variable contains an error object. Let's now create a new script, `ErrorVariable.ps1`, where we can play with error variables. Here is how we can get the variables and test for error conditions:

```

$files = dir z:\ -ErrorAction SilentlyContinue `
    -ErrorVariable errorVar
if ($errorVar)
{
    Write-Host "An error occurred getting files!"
}

del nosuchfile.txt -ErrorAction SilentlyContinue `
    -ErrorVariable errorVar
if ($errorVar)
{
    $realError = $errorVar[0]
    $realError | `
        Select FullyQualifiedErrorId,CategoryInfo,TargetObject | `
        Format-List

    $realError.InvocationInfo | `
        Select ScriptName,ScriptLineNumber,Line | Format-List
}

```

The `$errorVar` variable will hold a collection with more than one item if we got an error. We do not expect the first command to generate an error. In the second if statement, we get some of the properties for the error object and print them to the console using `Format-List`. So, this is what running the script looks like:

```
PS> .\ErrorVariable.ps1
```

```

FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.RemoveItemCommand
CategoryInfo          : ObjectNotFound: (Z:\09 - Error H...\nosuchfile.txt:String) [Remove-Item], ItemNotFoundException
TargetObject          : Z:\09 - Error Handling and Debugging\nosuchfile.txt

```

```

ScriptName            : Z:\09 - Error Handling and Debugging\errorvariable.ps1
ScriptLineNumber      : 7
Line                  : del nosuchfile.txt -ErrorAction SilentlyContinue -ErrorVariable e

```

The error object that we got back is a `System.Management.Automation.ErrorRecord` object. Our script outputs its most important properties. The `FullyQualifiedErrorId` is a unique identifier that we can use to verify that we have gotten exactly that error. We would not want to use string operations to detect that from the error message text, because that message will vary with localized versions of Windows. We can also get the error category, `ObjectNotFound` in our case. That may come in handy if we want to perform the same action on similar errors, that is, errors of the same category. The `TargetObject` property points to the object that we were working with at the time the error occurred; this is most likely the object that generated the error itself. If all else fails, we can turn to the `Exception` property and get the real .NET exception for our error record. Our script tries to go even further when it comes to verbosity and descriptiveness. It displays some of the properties of the error record `InvocationInfo` object. This object contains information about the execution context at the time of the error, and from it, we can get the actual script line that generated the error, its location in the script file, and the actual script file name. Neat, huh?

The `Verbose` and `Debug` cmdlet parameters are instructions that will generate additional information that will be printed on the console. Normally, that information is not printed so that it does not clutter the console window, but it may be useful when we are troubleshooting an error. Passing the `Verbose` and `Debug` switches to a command will make it generate that additional information. These parameters are intended primarily for script and cmdlet debugging, and we will look into them later when we get to the debugging part.

Trapping Errors

Calling actions with an `ErrorAction` of `SilentlyContinue` and then following that with an `if` statement to detect if an error has occurred is a good idea for one or two commands. It is a perfectly valid approach for that super-important command that is the backbone of your script. This approach has some shortcomings though: it is limited to operations involving cmdlets only and errors can get thrown by many other operations. Even when working with cmdlets, in many cases, we do not need to be as specific as catching errors one cmdlet at a time. One thing is certain—we would not want to litter all our cmdlet invocations with an extra parameter and an `if` statement just to be able to detect errors.

PowerShell solves this problem with its error trap language feature. A trap is an association between a block of code and a specific error. The code is registered in the current script file, function, or script block and is executed whenever the corresponding error occurs. Let's now use this feature to detect conversion errors that can be raised when trying to convert a string to a date. The `[datetime]` cast operator will raise such errors when the input string is not in the correct format. We can use a trap to write a script, `Date-Traps.ps1`, which goes over a collection of strings and converts them to dates. We know that PowerShell will raise a `System.Management.Automation.PSInvalidCastException` error, and we add a trap to handle that. Here is our script code:

```
trap [System.Management.Automation.PSInvalidCastException]
{
    Write-Host "Error converting a string to date!"
    $realException = $_.Exception.InnerException
    Write-Host "Inner Exception: $($realException.GetType())"
}
```

```
$dateStrings = "10/15/2007", "bad/date/format"
foreach ($dateString in $dateStrings)
{
    $date = [datetime] $dateString
    Write-Host $date
}
```

Since the trap statement accepts a script block, it complies with the convention of passing the current `ErrorRecord` in the `$_` variable. This time, we are interested in the `Exception` property. It contains the `PSInvalidCastException` that has been raised by the shell. We can reach deeper and get the actual .NET exception that was thrown by the .NET code used internally to actually parse the string and convert it to a date. This is available via the `InnerException` subproperty. This is what we get when we run our script:

```
PS> .\Date-Traps.ps1
10/15/2007 12:00:00 AM
Error converting a string to date!
Inner Exception: System.FormatException
Cannot convert value "bad/date/format" to type "System.DateTime". Error:
"The string was not recognized as a valid DateTime. There is a unknown
word starting at index 0."
At z:\09 - Error Handling and Debugging\Date-Traps.ps1:11 char:35
+     $date = [datetime] $dateString <<<<
```

The first date was converted successfully, and the second one was not. The error triggered, and our trap was executed showing the real .NET `System.FormatException`.

Suppressing Errors and Raising Them Again

The previous example handles an error, but it still shows the raw error message to the user. What if we want to hide that message? This is where it all gets tricky and counterintuitive—at least for those, like me, who are used to seeing the `break` and `continue` statements only in loops. To hide the error, we have to use a `continue` statement. This statement will suppress the error printout, and execution will continue with the *next statement* in the script. Let's now add another date to parse after the broken one; two diagnostic messages, one before and one after the loop; and a `continue` statement in the handler. The code now looks like this:

```
trap [System.Management.Automation.PSInvalidCastException]
{
    Write-Host "Error converting a string to date!"
    $realException = $_.Exception.InnerException
    Write-Host "Inner Exception: $($realException.GetType())"
    continue
}

Write-Host "script begin"
```

```

$dateStrings = "10/15/2007", "bad/date/format", "10/20/2007"
foreach ($dateString in $dateStrings)
{
    $date = [datetime] $dateString
    Write-Host $date
}

Write-Host "script end"

```

And here is what the output of our script looks like:

```

PS> .\Date-Traps.ps1
script begin
10/15/2007 12:00:00 AM
Error converting a string to date!
Inner Exception: System.FormatException
script end

```

We do not get the default error message anymore. Now, you can see why I emphasized the “next statement” phrase—the `continue` statement will exit the loop and continue with the first command after that. As you see, the “10/20/2007” string that we placed after the broken one did not get processed, because the loop is the operation that caused the error. The entire loop was terminated, and execution continued after that.

Suppressing an error and continuing execution is not the best recipe for every moment. Severe errors require some action on our part, such as cleaning up and notifying the user as well as stopping any further script execution. We can do that by using a `break` statement instead of a `continue` one. This is what our trap block looks like now:

```

trap [System.Management.Automation.PSInvalidCastException]
{
    Write-Host "Error converting a string to date!"
    $realException = $_.Exception.InnerException
    Write-Host "Inner Exception: $($realException.GetType())"
    break
}

```

That small change significantly alters the way our script behaves. Here is the script output:

```

PS> .\Date-Traps.ps1
script begin
10/15/2007 12:00:00 AM
Error converting a string to date!
Inner Exception: System.FormatException
Cannot convert value "bad/date/format" to type "System.DateTime". Error:
"The string was not recognized as a valid DateTime. There is a unknown
word starting at index 0."
At z:\09 - Error Handling and Debugging\Date-Traps.ps1:14 char:35
+     $date = [datetime] $dateString <<<<

```

We handled the error and wrote out our diagnostic messages in the trap handler. The error then got written to the console, and the script was terminated.

There is no limit to the number of trap statements that we can register, and this is how we handle different errors. To see that in action, we can show how to cover different conversion and arithmetic errors that can occur when working with numbers. Our script, `Number-Traps.ps1`, will try to convert a string to a number; that will fail because the string does not even look close to a number. The script will then perform an illegal division operation; it will attempt to divide by zero. Both errors will be handled in separate traps. Here is our script code:

```
Write-Host "script begin"

[int] "not a number"
$denominator = 0
$result = 50 / $denominator

Write-Host "script end"

trap [System.Management.Automation.PSInvalidCastException]
{
    Write-Host "Error converting a string to a number!"
    continue
}

trap [System.DivideByZeroException]
{
    Write-Host "Attempted to divide by zero!"
    continue
}
```

This time, I have placed the error traps at the end of the script. I did that to make a point: the location of the trap declarations does not matter. If we did not mind turning our script into an unreadable mess, we could have even interspersed the traps among the regular code lines. In addition, the order of defining our traps in the script code does not matter, because PowerShell parses the script and sets up all traps before starting execution. Most code on the Internet uses a convention of placing traps in the beginning of the code to which they apply, so sticking to that convention is best. Finally, here is what our script produces when we run it:

```
PS> .\Number-Traps.ps1
script begin
Error converting a string to a number!
Attempted to divide by zero!
script end
```

Catching Unknown Errors or All Errors

We do not always know the type of error that will be raised, or we may want to set up a trap for all errors that can be raised by a piece of code. To do that, we need to simply omit the type of the error in our trap declaration. The code in our error handler has to use the `$_` variable and the properties we showed before to distinguish between errors. Let's look at how to do that by rewriting our

previous example using a generic trap handler. We save the code to `Number-GenericTrap.ps1`, where we will use one trap handler that prints the `FullyQualifiedErrorId`, `Exception`, and `Exception.InnerException` properties. Here is the code:

```
trap
{
    $exceptionType = $_.Exception.GetType()
    $innerExceptionType = "No inner exception"
    if ($_.Exception.InnerException)
    {
        $innerExceptionType = $_.Exception.InnerException.GetType()
    }

    Write-Host "FullyQualifiedErrorId: $($_.FullyQualifiedErrorId)"
    Write-Host "Exception: $exceptionType"
    Write-Host "InnerException: $innerExceptionType"

    continue
}

Write-Host "script begin"

[int] "not a number"
$denominator = 0
$result = 50 / $denominator

Write-Host "script end"
```

This is what happens when we run the script:

```
PS> .\Number-GenericTrap.ps1
script begin
FullyQualifiedErrorId: RuntimeException
Exception: System.Management.Automation.PSInvalidCastException
InnerException: System.FormatException
FullyQualifiedErrorId: RuntimeException
Exception: System.DivideByZeroException
InnerException: No inner exception
script end
```

We get both errors—the `PSInvalidCastException` and the `DivideByZeroException`—just as we did in the previous versions of our script. Unfortunately, we do not get anything particularly meaningful in our `FullyQualifiedErrorId` property; both objects have the generic `RuntimeException` error ID, which is the error ID value for all exceptions coming out of the .NET runtime environment. To distinguish between the actual error types, we have to look at the `Exception` and `Exception.InnerException` objects. You just have to be careful, as the `InnerException` property for our exception is not guaranteed to contain a real object, as is the case with the `DivideByZeroException`.

Our generic trap handler has another good use. We can use it to get more information about an unknown error that we are getting. Suppose that we do not want to catch and suppress all errors, just one specific type of error. We do not know the exact type of the exception, and therefore, we cannot set up a specific trap handler. The solution to this problem is to set up a generic error trap, catch the error record object, and get the type of the object that the `Exception` property contains. This is our exception type that we can now use to write a specific trap handler and remove the generic one. The generic trap technique is very powerful—this is how I found out about the `PSInvalidCastException` and `DivideByZeroException` types in the examples shown so far.

Other programming languages provide constructs to mark a block of code and register traps for specific errors that may be raised by that code. PowerShell does not allow that—error traps are always registered for the current execution context. If we want to have a specific trap that is valid for a small piece of code, we need to move that code to a separate script block or a function. This way, we can set up nested traps: traps that take care of specific errors raised by a small piece of code can be moved inside functions and script blocks, and the more general handlers can be placed outside. If an error occurs inside a function, the shell will look for a trap that has been registered. If it does not find one, it will look for a trap in the parent scope and so on, until it reaches the script or global level. To demonstrate that, we will define a function, `Generate-DivideByZero`, in a new script file, `Nested-Traps.ps1` that will raise a `System.DivideByZeroException` error. This function will contain an error trap that will log a message to the console. In addition, the script generates a string-to-number conversion error that will be caught by a global error trap. Here is our code:

```
trap
{
    $error = $_.Exception.GetType()
    Write-Host "Caught $error outside function"
    continue
}

function Generate-DivideByZero
{
    trap
    {
        $error = $_.Exception.GetType()
        Write-Host "Caught $error inside function"
        continue
    }

    $denominator = 0
    $result = 50 / $denominator
}

Write-Host "script begin"

Generate-DivideByZero
[int] "not a number"
```

```
Write-Host "script end"
```

We should get two errors. Here is our script result:

```
PS> .\Nested-Traps.ps1
script begin
Caught System.DivideByZeroException inside function
Caught System.Management.Automation.PSInvalidCastException outside func
tion
script end
```

Low-level traps can pass the error to upper-level ones if needed. This incredibly useful feature allows us to create error handler chains that do something in response to an error and then pass the error up the chain for further processing. To pass up the error, we have to change the continue statement to a break one. Here is our modified script code:

```
trap
{
    $error = $_.Exception.GetType()
    Write-Host "Caught $error outside function"
    continue
}

function Generate-DivideByZero
{
    trap
    {
        $error = $_.Exception.GetType()
        Write-Host "Caught $error inside function"
        break
    }

    $denominator = 0
    $result = 50 / $denominator
}

Write-Host "script begin"

Generate-DivideByZero
[int] "not a number"

Write-Host "script end"
```

The output changes to this:

```
PS> .\Nested-Traps.ps1
script begin
Caught System.DivideByZeroException inside function
Caught System.DivideByZeroException outside function
Caught System.Management.Automation.PSInvalidCastException outside function
script end
```

The thing that is different than the previous time is that the `System.DivideByZeroException` error bubbled up to the global trap after being handled by the local one.

Last, but not least, we will discuss a useful practice that we can resort to whenever we publish a script for other people's use. Nobody is perfect, and having a good way to gather feedback about our script's problems is a real time saver. If users have an easy way to package an error report and send it back to us, we can spot and fix problems quickly. What can we do to have a log of any error's occurrences just as users have seen it on their machines? Even better—what can we do to get additional information that is of little use to the user running our script? We can get all this information by registering a global error trap in our script that will catch all errors and save them to a known location so that users can just e-mail the error dump back to us. Let's now pretend that we have a script called `Log-AllErrors.ps1` that tries to delete an important file. If the operation generates a terminating error, we would like to save that information and politely ask the user to send the file to us for further inspection. Here is the code:

```
trap
{
    $dumpFile = "error-dump.xml"
    $message = @"
A critical error occurred!
The error has been dumped to $dumpFile. Please include that file
in all error reports you send to scriptauthor@test.com
"@

    Write-Host $message -ForegroundColor Red
    $_ | Export-CliXml $dumpFile
    break
}

del veryimportantfile.txt -ErrorAction Stop
```

We use the `Export-CliXml` cmdlet to serialize the `ErrorRecord` object and turn it into an XML file. This way, we automatically get all the information and do not have to choose which error properties we need to save. In addition, we can create scripts that can process error reports automatically if, God forbid, we are swamped with many bug reports from our users. This is what we get when our script bombs at the console:

```
PS> .\Log-AllErrors.ps1
A critical error occurred!
The error has been dumped to error-dump.xml. Please include that file
in all error reports you send to scriptauthor@test.com
Remove-Item : Command execution stopped because the shell variable "ErrorActionPreference" is set to Stop: Cannot find path 'z:\09 - Error Handling and Debugging\veryimportantfile.txt' because it does not exist
.
At z:\09 - Error Handling and Debugging\Log-AllErrors.ps1:15 char:4
```

```
+ del <<<< veryimportantfile.txt -ErrorAction Stop
```

Our generated XML file looks roughly like this (I have cut out large portions for the sake of brevity):

```
<Objs Version="1.1"
  xmlns="http://schemas.microsoft.com/powershell/2004/04">
  <Obj RefId="RefId-0">
    <TN RefId="RefId-0">
      <T>System.Management.Automation.ErrorRecord</T>
      <T>System.Object</T>
    </TN>
    <MS>
      <Obj N="Exception" RefId="RefId-1">
        ...
      </Obj>
      ..
    </MS>
  </Obj>
</Objs>
```

The generated file is very readable, but it is best if we use the PowerShell tools to process it. We can read it back into an object using the `Import-CliXml` cmdlet:

```
PS> $e = Import-Clixml error-dump.xml
PS> $e.FullyQualifiedErrorId
ActionPreferenceStop,Microsoft.PowerShell.Commands.RemoveItemCommand
PS> $e.InvocationInfo

MyCommand      : Remove-Item
ScriptLineNumber : 15
OffsetInLine    : 4
ScriptName      : z:\09 - Error Handling and Debugging\Log-AllErrors.ps1
Line            : del veryimportantfile.txt -ErrorAction Stop
PositionMessage :
                  At z:\09 - Error Handling and Debugging\Log-AllErrors.ps1:15 char:4
                  + del <<<< veryimportantfile.txt -ErrorAction Stop

InvocationName  : del
PipelineLength  : 1
PipelinePosition : 1
```

We can get pretty detailed information, including the name of the script, the path it was run from, and the specific line that caused the error. Now, that is what I call a good error report!

Errors That Cannot Be Trapped

Some scripting languages are very liberal in their error-handling capabilities. Some even allow scripters to trap and suppress syntax errors. For good or bad, PowerShell will not allow us to do that. Syntax and parse errors are impossible to trap here. This feels like a good decision on the language designers' part, as there is little value in being able to suppress a syntax error. Almost all the time, a syntax error is a critical problem that needs fixing instead of suppressing or ignoring.

Parse errors should be easy to spot, right? Most of the time, a parse error is a piece of illegal code like an unrecognized statement an unbalanced set of braces, brackets, or quotation marks. There is one place where PowerShell stretches that definition a bit that needs mentioning—parse-time optimizations. The most common problem we can hit is literal expression evaluation. PowerShell tries to evaluate expressions like $3 + 4$ at parse time. This looks like a good optimization strategy, as it simplifies processing later on, but it has the shortcoming of not allowing us to trap any errors that occur during expression evaluation. Here is a sample script, `DivideByZero.ps1`, that tries to trap a `DivideByZeroException`:

```
trap
{
    Write-Host "Caught $($_.Exception.GetType())"
}

Write-Host "Script start"
3 / 0
```

Running the code gives us the standard error message:

```
PS> .\DivideByZero.ps1
Attempted to divide by zero.
At z:\09 - Error Handling and Debugging\DivideByZero.ps1:6 char:4
+ 3 / <<<< 0
```

Our trap has not triggered at all! As a matter of fact, no code has executed at all—we would have seen our “Script start” message if it had. PowerShell treats this situation as a parse error and just bails out. Compare the situation when we get rid of the `0` literal and replace it with something that gets evaluated to zero at runtime:

```
trap
{
    Write-Host "Caught $($_.Exception.GetType())"
    continue
}

Write-Host "Script start"
$zero = 0
3 / $zero
```

Here is the result:

```
PS> .\DivideByZero.ps1
Script start
Caught System.DivideByZeroException
```

As you can see, this time our trap got triggered, and we were able to suppress the error. The good news is that there are not many situations where you may hit a parse error that looks like a runtime one. If you encounter one of them, you now know how to verify that this is the case—add a simple `Write-Host` call in the beginning of the script. If it does not execute, you are dealing with a parse-time error.

Capturing Nonterminating Errors

Nonterminating errors do not stop script execution and cannot be caught with an error trap. To get hold of the `ErrorRecord` that has been generated for a nonterminating error, we have several options: use the `ErrorVariable` parameter to store the error to a location that we can later access; use error pipeline redirection; inspect the global `$Error` variable. We have already looked at the `ErrorVariable` option: it works, but it is pretty clumsy, as it requires that we pass an extra parameter to all cmdlets, and it still does not handle nonterminating errors raised by other code.

Redirecting the error pipeline is not really an error-capturing mechanism. Most often, it is used to simply suppress errors so that users do not see an error message at the console. Error redirection is similar to output redirection. For example, here is how we redirect the output of a command using the `>` operator:

```
PS> dir > directory-list.txt
PS> type directory-list.txt
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::z:\09 - Error Han
dling and Debugging
```

Mode	LastWriteTime	Length	Name
-a---	10/21/2007 2:26 PM	464	Date-Traps.ps1
-a---	10/22/2007 1:01 AM	788	directory-list.txt
...			

To redirect the error pipeline, we need to use the `2>` redirection operator. Here is how to do that in deleting a file that does not exist:

```
PS> del nosuchfile.txt 2> error.txt
PS> type error.txt
Remove-Item : Cannot find path 'z:\09 - Error Handling and Debugging\n
osuchfile.txt' because it does not exist.
At line:1 char:4
+ del <<<< nosuchfile.txt 2> error.txt
```

Most often, we would redirect to the `$null` variable so that we do not need to create files on disk:

```
PS> del nosuchfile.txt 2> $null
PS>
```

The 2 in the 2> operator is interesting from an archeological point of view. Console applications, by convention, write their output to the so-called `stdout` stream (the standard output), or the stream with an ID of 1. All error output is written to the `stderr` stream (the standard error), which has an ID of 2. With shells like `cmd.exe` and `bash`, the 2> operator redirects output written to `stderr` so that it is not shown to the screen but rather dumped in a file. The PowerShell designers have kept that syntax for redirecting the error pipeline, even though that shell uses object pipelines instead of text-based streams.

At the end of the day, the global `$Error` variable remains the only reliable way to get a nonterminating error. That variable is a collection that literally contains all errors that have occurred since the start of our shell session. Well, that should read “almost all errors”—the number is limited by the `$MaximumErrorCount` variable value, which defaults to 256. The `$Error` collection works a bit backward—it keeps the last error at the front, at `$Error[0]`. The error that occurred before that is at `$Error[1]` and so on. This is how we can get our `ErrorRecord` for the file delete operation that failed in the previous example:

```
PS> del nosuchfile.txt 2> $null
PS> $Error[0]
Remove-Item : Cannot find path 'z:\09 - Error Handling and Debugging\n
osuchfile.txt' because it does not exist.
At line:1 char:4
+ del <<<< nosuchfile.txt 2> $null
```

The `$Error` collection holds tremendous power. We can use it to build a logging facility that can keep track of all errors that have occurred between two points in time. All we have to do is to set a baseline by obtaining the number of errors stored in `$Error` and then get the number of errors again after we are finished. Having that information, we can traverse the collection and get the errors raised by the code executed between those two points. Let's now do that in a sample script, `Log-NonTerminatingErrors.ps1`, that will cause two errors and will suppress those using different techniques. Here is the code:

```
trap
{
    continue
}

Write-Host "setting error baseline"
$initialErrorCount = $error.count

$denominator = 0
4/$denominator

del nosuchfile.txt 2> $null

$errorCount = $error.count - $initialErrorCount
Write-Host "Errors since baseline"
for ($i = $errorCount - 1; $i -ge 0; $i--)
{
    $error[$i]
}
```

Dividing by zero is usually a terminating error, but we catch it with an error trap and just continue execution. The `del` operation fails, but its error pipeline is redirected to `$null`. The real error reporting mechanism obtains the initial error count. At the end, it gets the current error count and starts going over the errors backward, so that it can print them out in the chronological order of their occurrence. Here is what happens when we run our script:

```
PS> .\Log-NonTerminatingErrors.ps1
setting error baseline
Errors since baseline
Attempted to divide by zero.
At z:\09 - Error Handling and Debugging\Log-NonTerminatingErrors.ps1:1
0 char:3
+ 4/$ <<<< denominator
Remove-Item : Cannot find path 'z:\09 - Error Handling and Debugging\
osuchfile.txt' because it does not exist.
At z:\09 - Error Handling and Debugging\Log-NonTerminatingErrors.ps1:1
2 char:4
+ del <<<< nosuchfile.txt 2> $null
```

As we expected, we get the two errors and some details on what actually happened. This can be extended using ideas from the global trap technique to set a baseline at the beginning of the script and save all errors that have occurred during the entire script execution if a fatal error occurs. The actual implementation should be a hybrid between the preceding example and the global trap one, and I leave that as an exercise for you.

Raising Errors

Merely catching and handling errors would be too boring, because it would leave us on one side of the fence. Having suffered the pain of being thrown all sorts of errors, we would now like to be able to return the favor by throwing some errors ourselves. Jokes aside, too often, we have to provide a function or a ready script that will not be used directly by users but will be run by another script or program. Our script needs a way to indicate to its client code that it has encountered an abnormal condition and needs some action performed. “Abnormal condition” is a pretty loosely defined term and can mean anything: bad input data, a problem with the environment such as insufficient permissions to perform an operation, and so on. When we detect a condition like those, the most logical thing to do is to raise an error to the calling code and let it handle that. The caller can decide whether it should absorb the error and continue, change the input and try something else, or give up entirely and terminate execution.

Raising Terminating Errors

Terminating errors or exceptions are to be raised when something, well, exceptional happens and execution cannot continue. Suppose you are creating a script that knows how to process text files of a specific format. Someone ignores the instructions though and passes a `.ps1` file—a PowerShell script instead of the requisite text file. There is no way to get the job done, so we must throw an exception back. We can do that by using the `throw` statement.

To demonstrate how to throw errors, we will create a script, `Function-Parameters.ps1`, that will define a function `Find-TextFile`, which in turn requires a file name as a parameter.

The function will throw an error if it receives a name that does not have the .txt file extension. Here is the code:

```
function Find-TextFile($name)
{
    if ($name -notlike "*.txt")
    {
        throw "Find-TextFile: Expecting *.txt files only"
    }

    return Get-Item $name
}
```

```
Find-TextFile "Function-Parameters.ps1"
```

This is what happens when we run the script:

```
PS> .\Function-Parameters.ps1
Find-TextFile: Expecting *.txt files only
At z:\09 - Error Handling and Debugging\Function-Parameters.ps1:5 char
:14
+         throw <<<< "Find-TextFile: Expecting *.txt files only"
```

PowerShell is a lot different than the .NET programming languages like C# and VB.NET with regard to throwing exceptions. The .NET languages require that the object you throw be an object of a type that inherits from `System.Exception`. PowerShell does not have that restriction—in our example, we throw a simple string. In actuality, PowerShell too has that inheritance restriction, but it wraps the string in a `RuntimeException` object for us. Let's catch that exception and see for ourselves. After adding an error trap, the code looks like this:

```
trap
{
    Write-Host "$($_.Exception.GetType())"
    continue
}

function Find-TextFile($name)
{
    if ($name -notlike "*.txt")
    {
        throw "Find-TextFile: Expecting *.txt files only"
    }

    return Get-Item $name
}

Find-TextFile "Function-Parameters.ps1"
```

After running it, we get this:

```
PS> .\Function-Parameters.ps1
System.Management.Automation.RuntimeException
```

Throwing all types of objects at our clients is convenient for us but not very convenient for the client code. For one thing, it is impossible to write a specific error trap, and we have to resort to heuristics to identify one error from another. Life would be a mess if everyone threw only strings as exceptions. There has to be a better way!

The way out of this mess is to create an exception object and throw that. Let's rework our code so that it throws a `System.ArgumentException` instead:

```
trap [System.ArgumentException]
{
    Write-Host "$($_.Exception.Message)"
    continue
}

function Find-TextFile($name)
{
    if ($name -notlike "*.txt")
    {
        $message = "Find-TextFile: Expecting *.txt files only"
        throw (New-Object System.ArgumentException -arg $message)
    }

    return Get-Item $name
}

Find-TextFile "Function-Parameters.ps1"
```

Note that we are now using the `New-Object` cmdlet to create a .NET `System.ArgumentException` object and pass the error message as a constructor parameter. This allows us to write a more specific error trap that will catch only argument exceptions. Here is the result that we get after running our script:

```
PS> .\Function-Parameters.ps1
Find-TextFile: Expecting *.txt files only
```

In Chapter 5, I showed you how to use the `throw` statement inside an expression that is specified as a function parameter's default value to raise an error if the parameter has not been supplied. That clever hack to implement mandatory parameters can be further extended with standard .NET exception objects. To continue our example, we can add a check that the `$name` parameter is not `$null`. Here is the new version of our script:

```
trap [System.ArgumentException]
{
    Write-Host "$($_.Exception.Message)"
    continue
}
```

```
function Find-TextFile($name = `
    $(throw (New-Object System.ArgumentNullException -arg "name")))
{
    if ($name -notlike "*.txt")
    {
        $message = "Find-TextFile: Expecting *.txt files only"
        throw (New-Object System.ArgumentException -arg $message)
    }

    return Get-Item $name
}

Find-TextFile "Function-Parameters.ps1"
Find-TextFile
```

In the new function definition, we use a somewhat convoluted parameter default value expression that creates a new `System.ArgumentNullException` for the `$name` argument and throws it if the argument has not been supplied. Here, we enjoy the benefits of object-oriented programming to the fullest—the `ArgumentNullException` type in .NET inherits from the `ArgumentException` one. That means that all `ArgumentNullException` objects are `ArgumentException` ones too. That, in turn, means that traps set up for `ArgumentException` errors catch all errors of derived types. Here is what the script output looks like:

```
PS> .\Function-Parameters.ps1
Find-TextFile: Expecting *.txt files only
Value cannot be null.
Parameter name: name
```

Note that we get two different errors for the two invocations. The use of `ArgumentNullException` shines with a nicely formatted error message about a missing parameter. Use that exception for missing parameters; it will provide the best user experience, and it will even contain localized error messages for non-English Windows systems.

Raising Nonterminating Errors

Not all errors that we encounter are fatal for script execution. We may be able to continue with our logic, but we may still want to issue a mild warning to the user or to the client code. The information in that warning can help the client code make an intelligent decision on how to continue. Even if the client code ignores it, the information in the nonterminating error message is likely to reach the user, and that can help him or her in diagnosing and fixing a problem.

So, how do we generate a nonterminating error? There is no support built into the PowerShell language to do that, but fortunately, we have the `Write-Error` cmdlet. That cmdlet can take a message string, an exception, or an `ErrorRecord` object and pass it down the error pipeline. To see that in action, let's go back to finding text files. Our new script, `Function-ParametersNonTerminating.ps1`, will define a function `Find-TextFile` that will return the item for the given path and generate a nonterminating error if the file does not exist. Here is the first version of the script:

```
function Find-TextFile($name)
{
    if (!(Test-Path $name))
    {
        Write-Error -Message "$name does not exist"
        return $null
    }
    else
    {
        return Get-Item $name
    }
}
```

```
Write-Host "Script start"
Find-TextFile "nosuchfile.txt"
Write-Host "Script end"
```

Invoking the script prints the error in red but does not stop execution when the error occurs:

```
PS> .\Function-ParametersNonTerminating.ps1
Script start
Find-TextFile : nosuchfile.txt does not exist
At z:\09 - Error Handling and Debugging\Function-ParametersNonTerminat
ing.ps1:15 char:14
+ Find-TextFile <<<< "nosuchfile.txt"
Script end
```

Since `Write-Error` writes to the error pipeline, the client code can easily suppress the error by redirecting that pipeline to `$null`:

```
PS> .\Function-ParametersNonTerminating.ps1 2> $null
Script start
Script end
```

The script looks perfect, but once we start poking around, we find some things that can easily be improved. The most annoying thing is that when we use the `Message` parameter, we always get the same stock exception type, `WriteErrorException`:

```
PS> .\Function-ParametersNonTerminating.ps1
Script start
Find-TextFile : nosuchfile.txt does not exist
At z:\09 - Error Handling and Debugging\Function-ParametersNonTerminat
ing.ps1:15 char:14
+ Find-TextFile <<<< "nosuchfile.txt"
Script end
PS> $Error[0].Exception.GetType().FullName
Microsoft.PowerShell.Commands.WriteErrorException
```

Again, we would have to heuristically inspect the string contents of error messages to detect what really happened. We would not want to do that to our client code, right? The best

thing to do here is to create a `System.IO.FileNotFoundException` object, initialize it with the correct error message and file name, and pass it to `Write-Error`. Here is how to modify our code to make it do that:

```
function Find-TextFile($name)
{
    if (!(Test-Path $name))
    {
        $errorMessage = "$name does not exist"
        $exception = New-Object System.IO.FileNotFoundException `
            -arg $message,$name
        Write-Error -Exception $exception
        return $null
    }
    else
    {
        return Get-Item $name
    }
}

Write-Host "Script start"
Find-TextFile "nosuchfile.txt"
Write-Host "Script end"
```

The code is not much different, other than the actual exception object creation line and the new method of calling `Write-Error` through its `Exception` parameter. This is what happens when we run the script and inspect the last error:

```
PS> .\Function-ParametersNonTerminating.ps1
Script start
Find-TextFile :
At z:\09 - Error Handling and Debugging\Function-ParametersNonTerminat
ing.ps1:18 char:14
+ Find-TextFile <<<< "nosuchfile.txt"
Script end
PS> $Error[0].Exception.GetType().FullName
System.IO.FileNotFoundException
```

Note that the text message that got written to the console has not changed at all. What really matters has changed, though—our exception is now of the correct type, and that makes the client code's task of finding out what type of error occurred a whole lot easier.

Debugging Your Code

Debugging is the black art of finding and eliminating errors in computer programs. Logic errors have long been referred to as bugs, and popular legend has it that the first bug was a real moth stuck between the contacts of a relay in the U.S. Navy's Harvard Mark II computer. The fix back then was simple; the bug was extracted using tweezers.

Most programming languages and environments ship with a fully featured debugger system, a means to execute a program step-by-step and to inspect the intermediate results of its execution. For better or for worse, PowerShell does not contain a script debugger, and we have to resort to other techniques to eliminate errors in our programs. Luckily, using some ingenious tricks, we can get pretty close to tracing our program execution, temporarily suspending execution flow and examining the state of the program.

As with any complex task, we do not need super-fancy tools to debug our scripts. The most important part of tackling a misbehaving program is to be able to divide the problem into small subproblems and conquer them all one by one. Just as prophylaxis is the best cure in medicine, with debugging, the best practice is bug prevention. The key to squashing any bug is to structure your script in small blocks that can be tested in isolation. This way, you are guaranteed to encounter only tiny bugs—the kind that are easy to kill. Whenever you feel that you do not understand what is really going on in your program, first turn to Chapters 4, 5, and 8 to review the techniques that you can employ to simplify your program.

Print Debugging

Print debugging is the most primitive and quite often the most effective form of debugging. The idea is simple—we inspect the state of the program at a specific point in the execution flow by injecting code that prints information about the system. The real power of this technique lies in the fact that we have the full power of the programming language to get to any part of the system and print information about it: script variables, environment variables, file contents, registry keys, and so on. That power can get dangerous; it is too easy to get carried away and introduce a logic error in a routine that is supposed to just print information, and that alone is enough to poison our entire debugging session. The key to using this technique effectively is to keep the injected print statements simple.

The most convenient cmdlet that we can use to print debug strings is `Write-Host`. It does not output objects or strings to the object pipeline but writes them directly to the console. We have been using that cmdlet to provide diagnostic messages that do not interfere with the current pipeline throughout this entire book. In addition to being able to print out all sorts of objects and collections, `Write-Host` can use color to make its output even more beautiful. We can use that to color code our messages and provide additional cues to the user by, say, coloring diagnostic messages in green, warnings in yellow, and errors in red. We can do that using the `BackgroundColor` and `ForegroundColor` parameters. Here are the possible values:

```
PS> $helpTopic = (Get-Help Write-Host -Parameter ForegroundColor)
PS> $helpTopic.possibleValues.possibleValue | select Value
```

```
value
-----
Black
DarkBlue
DarkGreen
```

DarkCyan
DarkRed
DarkMagenta
DarkYellow
Gray
DarkGray
Blue
Green
Cyan
Red
Magenta
Yellow
White

The possible `BackgroundColor` values are the same. In the preceding command, we are using the fact that even help topics in PowerShell are objects, and we can manipulate them with standard cmdlets. To learn more about similar techniques for obtaining help for a command, please see Chapter 13.

Let's now use some color to create a small script called `Print-Debug.ps1` that will count the number of characters in a text file. We will define the `Count-Characters` function and use `Write-Host` inside to write diagnostic messages. Here is the code:

```
function Count-Characters($file)
{
    Write-Host "Opening $file" -Background White -Foreground DarkGreen
    $content = ""
    if (Test-Path $file)
    {
        $content = Get-Content $file
    }

    Write-Host "File contains $($content.Length) characters" `
        -Background White -Foreground DarkGreen
}
```

```
Count-Characters Print-Debug.ps1
Count-Characters nosuchfile.txt
```

Our function defaults to an empty content string. It checks if the file exists using the `Test-Path` cmdlet, and if that is the case, it reads the file contents using the `Get-Content` cmdlet. The function uses dark green text on a white background to write two diagnostic messages: one before it opens the file and another when it finishes reading the file contents. Figure 9-1 shows a screenshot of the console window.

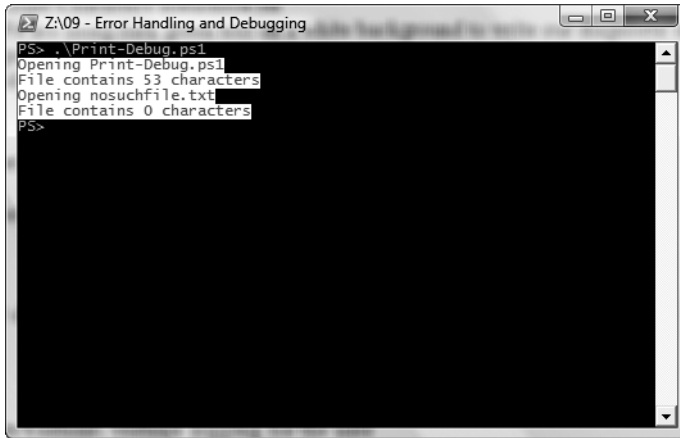


Figure 9-1. Here's the output generated by the `Write-Host` cmdlet; onscreen, the text is dark green on a white background.

Generating Verbose Output

Quite often, we need diagnostic messages that the user does not need to see. We may require quite verbose output when we are developing the script or troubleshooting a problem. That output has no value for the user; on the contrary, it clutters the console screen and can be harmful. We need a way to generate diagnostic messages that we can easily turn on and off. PowerShell supports that need quite nicely with its diagnostic cmdlets: `Write-Verbose`, `Write-Warning`, and `Write-Debug`. Here is what happens when we invoke `Write-Verbose` to write a message that should be seen only when the user has explicitly specified so:

```
PS> Write-Verbose "This is a verbose message"
PS>
```

That is right! There is no output at all. We need to specify that we are interested in verbose output first. We can do that by providing the `Verbose` common parameter:

```
PS> Write-Verbose "This is a verbose message" -Verbose
VERBOSE: This is a verbose message
```

The `Verbose` parameter still does not allow us to switch verbose output generation on and off easily. That is why PowerShell provides another mechanism to control that—the global `$VerbosePreference` variable. That variable contains an `ActionPreference` object, the same object that we provided as the `ErrorAction` common parameter in the beginning of this chapter. The default value for `$VerbosePreference` is `SilentlyContinue`, and that is why we do not see any output. To make our output visible, we need to set it to `Continue`:

```
PS> $VerbosePreference = "Continue"
PS> Write-Verbose "This is a verbose message"
VERBOSE: This is a verbose message
```

We can also set `$VerbosePreference` to `Stop` if we want the shell to generate a stop error that will prevent the rest of the code from executing. The `Inquire` option is supported too; if turned on, the shell will ask the user what to do next:


```
PS> $VerbosePreference = "Inquire"
PS> Write-Verbose "This is a verbose message"
VERBOSE: This is a verbose message
```

Confirm

Continue with this operation?

```
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help
(default is "Y"):
```

Probably the best use for `Write-Verbose` is to generate messages that can be seen by advanced users only at times when they need to troubleshoot a problem. That is why the most useful values for `$VerbosePreference` are the ones that turn output on and off or `Continue` and `SilentlyContinue`.

We can now modify our `Count-Characters` function in such a way so that it generates verbose messages right after it is called and immediately before it finishes execution. Here is what our script file looks like now:

```
function Count-Characters($file)
{
    $functionName = $MyInvocation.MyCommand.Name
    Write-Verbose "Entering $functionName"

    Write-Host "Opening $file" -Background White -Foreground DarkGreen
    $content = ""
    if (Test-Path $file)
    {
        $content = Get-Content $file
    }

    Write-Host "File contains $($content.Length) characters" `
        -Background White -Foreground DarkGreen

    Write-Verbose "Leaving $functionName"
}
```

```
Count-Characters Print-Debug.ps1
Count-Characters nosuchfile.txt
```

We get the function name from the `$MyInvocation` special variable to avoid hard-coding the name in our script. This way, we can copy and paste that code to different functions. Here is the output our script produces with verbose output turned off and on:

```
PS> $VerbosePreference = "SilentlyContinue"
PS> .\Print-Debug.ps1
Opening Print-Debug.ps1
File contains 58 characters
Opening nosuchfile.txt
File contains 0 characters
```

```

PS> $VerbosePreference = "Continue"
PS> .\Print-Debug.ps1
VERBOSE: Entering Count-Characters
Opening Print-Debug.ps1
File contains 37 characters
VERBOSE: Leaving Count-Characters
VERBOSE: Entering Count-Characters
Opening nosuchfile.txt
File contains 0 characters
VERBOSE: Leaving Count-Characters

```

Diagnostic messages are good, but we cannot afford to paste that ugly code in all our functions. We need a more elegant solution here. We can define a function called `Instrument-Function` that will use a script block to wrap the original function body. `Instrument-Function` will contain the diagnostics code; it will generate the verbose output before and after calling the original function body, and verbose messages will no longer litter our original code. Here is the modified code:

```

function Instrument-Function($body)
{
    $parentInvocation = Get-Variable MyInvocation -scope 1 -ValueOnly
    $functionName = $parentInvocation.MyCommand.Name

    Write-Verbose "Entering $functionName"
    &$body
    Write-Verbose "Leaving $functionName"
}

function Count-Characters($file)
{
    Instrument-Function {
        Write-Host "Opening $file" `
            -Background White -Foreground DarkGreen
        $content = ""
        if (Test-Path $file)
        {
            $content = Get-Content $file
        }

        Write-Host "File contains $($content.Length) characters" `
            -Background White -Foreground DarkGreen
    }
}

Count-Characters Print-Debug.ps1
Count-Characters nosuchfile.txt

```

As you can see, `Count-Character`'s definition looks almost the same as the original one. The only difference is that the original code is wrapped in a script block and passed to `Instrument-Function`. `Instrument-Function` needs the name of the function that called it to generate its diagnostic output. It does that by going the parent scope and getting the parent invocation info using the `Get-Variable` cmdlet. The code instrumentation approach to generating verbose output is so elegant that you can use it in all your functions. There is one small caveat: the new script block will not contain the original automatic `$Args` variable. That type of instrumentation will not work with unnamed function parameters.

Generating Debug Output

"Debug output" is just a fancy term for diagnostic messages generated by our script. The difference from verbose output is that debugging messages are only useful to the script author. Even advanced users able to troubleshoot problems on their own are unlikely to benefit from such messages, as debug output typically contains internal error messages, intermediate variable contents, and other information that can help the script author easily spot bugs and all other types of anomalies.

We can generate debug messages using the `Write-Debug` cmdlet. Here is our first message:

```
PS> Write-Debug "This is a debug message"
PS>
```

Oops! Debug output, just like verbose output, is disabled by default. To see something, we need to pass the `Debug` parameter:

```
PS> Write-Debug "This is a debug message" -Debug
DEBUG: This is a debug message
```

```
Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):
```

The shell sees that we have a debug message coming, outputs it, and prompts us for action. Debug messages are controlled by a global shell variable too—`$DebugPreference`. Just like `$VerbosePreference`, this variable contains an `ActionPreference` object. Again, the default value is `SilentlyContinue`. We got the prompt in the previous example, because `Write-Debug` switches to `Inquire` when given the `Debug` switch. We can change the action preference to `Continue` to get a simple message:

```
PS> $DebugPreference = "Continue"
PS> Write-Debug "This is a debug message"
DEBUG: This is a debug message
```

Now that we have our `Instrument-Function` in place, we can extend it so that it traps all errors that have been raised by the function body. When we get an error, we will write a debug message and let it bubble up to other traps. To see that in action, we have to modify the `Count-Characters` function and make it throw an exception if the file does not exist. This is how our modified script looks:

```

function Instrument-Function($body)
{
    $parentInvocation = Get-Variable MyInvocation -scope 1 -ValueOnly
    $functionName = $parentInvocation.MyCommand.Name

    trap
    {
        Write-Debug "$functionName raised an error"
    }

    Write-Verbose "Entering $functionName"
    &$body
    Write-Verbose "Leaving $functionName"
}

function Count-Characters($file)
{
    Instrument-Function {
        Write-Host "Opening $file" `
            -Background White -Foreground DarkGreen
        $content = ""
        if (Test-Path $file)
        {
            $content = Get-Content $file
        }
        else
        {
            throw "No such file"
        }

        Write-Host "File contains $($content.Length) characters" `
            -Background White -Foreground DarkGreen
    }
}

```

```

Count-Characters Print-Debug.ps1
Count-Characters nosuchfile.txt

```

We have added a trap handler to Instrument-Function and an else clause in Count-Characters. Running the script with debug and verbose output enabled looks like this:

```

PS> $VerbosePreference = "Continue"
PS> $DebugPreference = "Continue"
PS> .\Print-Debug.ps1
VERBOSE: Entering Count-Characters
Opening Print-Debug.ps1
File contains 37 characters
VERBOSE: Leaving Count-Characters

```

```

VERBOSE: Entering Count-Characters
Opening nosuchfile.txt
DEBUG: Count-Characters raised an error
No such file
At Z:\09 - Error Handling and Debugging\Print-Debug.ps1:28 char:18
+         throw <<<< "No such file"
VERBOSE: Leaving Count-Characters

```

We can clearly see that the second function invocation entered the function but never left it properly. That happened because the code triggered an exception. We can go further and set `$DebugPreference` to `Inquire` so that we can suspend script execution and look around. This is useful to get additional information about the error. Here is how to do that:

```

PS> $DebugPreference = "Inquire"
PS> .\Print-Debug.ps1
VERBOSE: Entering Count-Characters
Opening Print-Debug.ps1
File contains 37 characters
VERBOSE: Leaving Count-Characters
VERBOSE: Entering Count-Characters
Opening nosuchfile.txt
DEBUG: Count-Characters raised an error

Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help
(default is "Y"):S
PS> $file
nosuchfile.txt
PS> exit

```

```

Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help
(default is "Y"):Y
No such file
At Z:\09 - Error Handling and Debugging\Print-Debug.ps1:28 char:18
+         throw <<<< "No such file"
VERBOSE: Leaving Count-Characters

```

We entered a fully functional command prompt and were able to get the value of the `$file` parameter. This showed us that the second invocation—the one trying to open `nosuchfile.txt`—actually caused the exception. We exited the nested command prompt with the `exit` command and returned to our `Write-Debug` inquiry. This is a very useful technique, and you can use it similar to a debugger breakpoint; your script execution is suspended, and you can look around and inspect variables' contents.

Generating Warnings

Warnings are the third kind of diagnostic output that we will discuss. They are typically used to output noncritical messages that users should see. Warning messages usually mean that something went wrong, but the script probably knows how to handle the situation. The information is presented to the user, so that he or she knows what really happened and can react accordingly if the script fails to handle things and crashes later.

We can output warning messages with the `Write-Warning` cmdlet. Here is how:

```
PS> Write-Warning "This is a warning"
WARNING: This is a warning
```

Warnings get output to the console by default. They too can be controlled via a global variable—`$WarningPreference`. That variable defaults to `Continue`, and we can change it to suppress warnings by setting `$WarningPreference` to `SilentlyContinue`. We can also ask the user for action by setting the variable to `Inquire` and possibly allow him or her to suspend execution and look around from a nested prompt.

That exception we threw in our `Count-Characters` function does not feel quite right. A value of zero looks a good default for nonexistent files, and we can change the exception to a warning. Here is the new version of our script:

```
function Instrument-Function($body)
{
    $parentInvocation = Get-Variable MyInvocation -scope 1 -ValueOnly
    $functionName = $parentInvocation.MyCommand.Name

    trap
    {
        Write-Debug "$functionName raised an error"
    }

    Write-Verbose "Entering $functionName"
    &$body
    Write-Verbose "Leaving $functionName"
}

function Count-Characters($file)
{
    Instrument-Function {
        Write-Host "Opening $file" `
            -Background White -Foreground DarkGreen
        $content = ""
        if (Test-Path $file)
        {
            $content = Get-Content $file
        }
    }
```

```

    else
    {
        Write-Warning "$file does not exist."
    }

    Write-Host "File contains $($content.Length) characters" `
        -Background White -ForegroundColor DarkGreen
    }
}

```

```

Count-Characters Print-Debug.ps1
Count-Characters nosuchfile.txt

```

Running the script twice: with verbose output turned on and then off, we get this:

```

PS> $VerbosePreference = "Continue"
PS> .\Print-Debug.ps1
VERBOSE: Entering Count-Characters
Opening Print-Debug.ps1
File contains 37 characters
VERBOSE: Leaving Count-Characters
VERBOSE: Entering Count-Characters
Opening nosuchfile.txt
WARNING: nosuchfile.txt does not exist.
File contains 0 characters
VERBOSE: Leaving Count-Characters
PS>
PS> $VerbosePreference = "SilentlyContinue"
PS> .\Print-Debug.ps1
Opening Print-Debug.ps1
File contains 37 characters
Opening nosuchfile.txt
WARNING: nosuchfile.txt does not exist.
File contains 0 characters

```

We are now confident that users will get a reasonable default when their file does not exist, but they will be notified of what really happened so that they do not rely on the file's existence later on. In addition, they can suppress the warning if they are not interested in it.

Controlling Error Output

Write-Verbose, Write-Debug, and Write-Warning all work in pretty much the same way. We can say the same for Write-Error too. It is not an output cmdlet like the others, but it is similar to the others in the sense that its output can be controlled via a well known global variable, `$ErrorActionPreference`. Now, you know yet another way to suppress error messages:

```

PS> $ErrorActionPreference = "SilentlyContinue"
PS> Write-Error "An error occurred"
PS>

```

Alternatively, we can set that variable to `Inquire` to conveniently break into a nested prompt whenever an unexpected error occurs.

Stepping Through Scripts and Breaking Execution

Stepping through a script is a special mode of execution that executes a script line by line and asks before continuing with the next line. It might seem boring at first, but it is an excellent way to trace what a script really does. Until now, you have seen ways to ask for user input before continuing execution for a single command originated from cmdlets like `Write-Debug`. We can do that for every statement by using the `Set-PSDebug` cmdlet. Let's go back to the simple version of our `Count-Characters` function, saved in the `Count-Characters.ps1` script file. Here is the code again:

```
function Count-Characters($file)
{
    $content = ""
    if (Test-Path $file)
    {
        $content = Get-Content $file
    }

    Write-Host "File contains $($content.Length) characters"
}
```

```
Count-Characters Print-Debug.ps1
Count-Characters nosuchfile.txt
```

To step through the script one statement at a time, we need to call `Set-PSDebug` with the `Step` parameter:

```
PS> Set-PSDebug -step
PS> .\Count-Characters.ps1
```

Continue with this operation?

```
1+ .\Count-Characters.ps1
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):Y
DEBUG: 1+ .\Count-Characters.ps1
```

Continue with this operation?

```
1+ function Count-Characters($file)
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):Y
```

We can suspend execution anytime and find ourselves in a nested prompt that we can use to get additional information about the environment. Again, when we are done, we can exit the prompt with the `exit` command. Let's continue our script stepping session:


```
DEBUG: 12+ Count-Characters Print-Debug.ps1
```

```
Continue with this operation?
```

```
3+ $content = ""
```

```
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"):S
```

```
PS> $file
```

```
Print-Debug.ps1
```

```
PS> exit
```

```
Continue with this operation?
```

```
3+ $content = ""
```

```
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"):
```

To restore the normal shell operating mode, either restart your shell session or invoke `Set-PSDebug` passing the `Off` parameter:

```
PS> Set-PSDebug -Off
```

```
Continue with this operation?
```

```
1+ Set-PSDebug -off
```

```
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"):
```

```
DEBUG: 1+ Set-PSDebug -Off
```

Simulating Breakpoints

Stepping through a large script can easily bore you to death—that is why modern debuggers allow you to set a breakpoint at a specific location. You can then execute the code, and it will run normally until it reaches that location. At that time, execution will be suspended, allowing you to inspect variable contents, step through the rest of the script, and much more. PowerShell does not support breakpoints, but we can at least simulate setting breakpoints by suspending script execution and starting up a nested prompt that will allow us to read variable contents and even modify the environment. The technique was first outlined on the PowerShell team blog, and the idea is to use the `EnterNestedPrompt` method that the host object must implement. This allows us to write a function that will suspend execution when it is called. Let's add that method to our `Count-Characters.ps1` script and call it when the file we are trying to open does not exist. Here is the code:

```
function Start-Debug
{
    Write-Host "Breakpoing hit!" -ForegroundColor Red
    function prompt
    {
        "DEBUG> "
    }
    $host.EnterNestedPrompt()
}
```

```
function Count-Characters($file)
{
    $content = ""
    if (Test-Path $file)
    {
        $content = Get-Content $file
    }
    else
    {
        Start-Debug

        Write-Host "File contains $($content.Length) characters"
    }
}
```

```
Count-Characters Print-Debug.ps1
```

```
Count-Characters nosuchfile.txt
```

We write a notification in red, so that it catches our attention, and change the prompt to make it immediately visible that we are in a debugging nested prompt before actually entering the prompt. This is how we can find out the name of the file that does not really exist:

```
PS> .\Count-Characters.ps1
File contains 37 characters
Breakpoint hit!
DEBUG> $file
nosuchfile.txt
DEBUG> exit
File contains 0 characters
```

Tracing Script Execution Details

One of PowerShell's greatest strengths with regard to error diagnostics and resolution is its support for tracing command execution. All operations are built with the idea in mind that a person—whether a script developer, a cmdlet developer, or an advanced user—should be able to get a trace log with the detailed operations that the shell itself or one of its cmdlets is performing. We will look into two types of tracing here: intrinsic shell operations, like setting variables and calling functions, and specific operations for which we can request tracing.

Tracing Intrinsic Shell Operations

Let's take a look at the simple task of logging a script's execution to determine the order in which operations were called: we need information on what operations were performed and in what order. To get that, we need to switch the shell debug tracing, which we do by calling the `Set-PSDebug` cmdlet and passing the `Trace` parameter. PowerShell offers two levels of tracing, 1 and 2, with the latter generating more verbose output. Let's create a small script, `Trace-Command.ps1`, that defines a function `Calculate` that divides two numbers. Here is the code:

```
function Calculate($a, $b)
{
    $result = $a / $b
    return $result
}
```

Calculate 1 2

To get the order in which the script lines were executed, we need to switch to trace level 1:

```
PS> Set-PSDebug -Trace 1
PS> .\Trace-Execution.ps1
DEBUG: 1+ .\Trace-Execution.ps1
DEBUG: 1+ function Calculate($a, $b)
DEBUG: 7+ Calculate 1 2
DEBUG: 3+     $result = $a / $b
DEBUG: 4+     return $result
0.5
```

You can see that line 1 was executed first, and it defined the Calculate function. Later, line 7 invoked that function. After that, execution went through the function body: lines 3 and 4. Let's see what happens when we enable trace level 2:

```
PS> Set-PSDebug -Trace 2
DEBUG: 1+ Set-PSDebug -Trace 2
PS> .\Trace-Execution.ps1
DEBUG: 1+ .\Trace-Execution.ps1
DEBUG: ! CALL script 'Trace-Execution.ps1'
DEBUG: 1+ function Calculate($a, $b)
DEBUG: 7+ Calculate 1 2
DEBUG: ! CALL function 'Calculate' (defined in file 'Z:\09 -
Error Handling and Debugging\Trace-Execution.ps1')
DEBUG: 3+     $result = $a / $b
DEBUG: ! SET $result = '0.5'.
DEBUG: 4+     return $result
0.5
```

In addition to the line numbers and the execution log, we get information about scripts and functions that have been called. We get a log for variable assignment; that is what the ! SET line means. Note that we get the computed value that resulted from evaluating the expression, which can provide invaluable information when debugging complex expressions. The function call log is very useful, as it shows the script that defined that function. We can use that information to troubleshoot function name collisions and unwanted function overwrites.

Trace level 2 provides a lot of internal information that we can use to learn how cmdlets work. Here is how we can get information about what Get-ChildItem does when listing a directory entry:

```

PS> Set-PSDebug -Trace 2
DEBUG: 1+ Set-PSDebug -Trace 2
PS> Get-ChildItem Trace-Execution.ps1
DEBUG: 1+ Get-ChildItem Trace-Execution.ps1
DEBUG: 1+ $_.PSParentPath
DEBUG: 1+ $catr = "";
DEBUG: ! SET $catr = ''.
DEBUG: 2+ if ( $this.Attributes -band 16 ) {
$catr += "d" } else { $catr += "-" } ;
DEBUG: 2+ if ( $this.Attributes -band 16 ) {
$catr += "d" } else { $catr += "-" } ;
DEBUG: ! SET $catr = '-'.
DEBUG: 3+ if ( $this.Attributes -band 32 ) {
$catr += "a" } else { $catr += "-" } ;
DEBUG: 3+ if ( $this.Attributes -band 32 ) {
$catr += "a" } else { $catr += "-" } ;
DEBUG: ! SET $catr = '-a'.
DEBUG: 4+ if ( $this.Attributes -band 1 ) {
$catr += "r" } else { $catr += "-" } ;
DEBUG: 4+ if ( $this.Attributes -band 1 ) {
$catr += "r" } else { $catr += "-" } ;
DEBUG: ! SET $catr = '-a-'.
DEBUG: 5+ if ( $this.Attributes -band 2 ) {
$catr += "h" } else { $catr += "-" } ;
DEBUG: 5+ if ( $this.Attributes -band 2 ) {
$catr += "h" } else { $catr += "-" } ;
DEBUG: ! SET $catr = '-a--'.
DEBUG: 6+ if ( $this.Attributes -band 4 ) {
$catr += "s" } else { $catr += "-" } ;
DEBUG: 6+ if ( $this.Attributes -band 4 ) {
$catr += "s" } else { $catr += "-" } ;
DEBUG: ! SET $catr = '-a---'.
DEBUG: 7+ $catr
DEBUG: 2+
[String]::Format("{0,10} {1,8}", $_.LastWriteTime.ToString("d"),
$_.LastWriteTime.ToString("t"))
DEBUG: ! CALL method 'System.String ToString(String format)'
DEBUG: ! CALL method 'System.String ToString(String format)'
DEBUG: ! CALL method 'static System.String Format(String format,
Object arg0, Object arg1)'

```

Directory: Microsoft.PowerShell.Core\FileSystem::Z:\09 - Error Handling and Debugging

Mode	LastWriteTime	Length	Name
----	-----	-----	
-a---	10/24/2007 7:56 AM	94	Trace-Execution.ps1

We get quite a big log that tells us that `Get-ChildItem` triggers some complex calculations related to file access attributes in order to output the file `Mode` string. It also gets the `LastWriteTime` and converts everything to strings by using the `System.String.Format` .NET method. As usual, to restore the normal shell operating mode, either restart your shell session or invoke `Set-PSDebug` passing the `Off` parameter.

Operation-Specific Tracing

PowerShell allows us to trace operations specific to one of its components. We can get information about command discovery or about what command is really invoked when we type something at the prompt, cmdlet parameter binding or which value went to what parameter, both implicit and explicit type conversions, and much more. To get that information, we have to use the `Trace-Command` cmdlet. It takes a script block that contains the actual command that needs tracing. The other important parameter is the component name.

Let's first use `Trace-Command` to get information about what command is really invoked when typing `gal ii` at the command line. We need tracing for the `CommandDiscovery` component. Here is how to do that:

```
PS> Trace-Command -Name CommandDiscovery -PSHost { gal ii }
DEBUG: CommandDiscovery Information: 0 : Looking up command: gal
DEBUG: CommandDiscovery Information: 0 : Alias found: gal Get-Alias
DEBUG: CommandDiscovery Information: 0 : Attempting to resolve
function or filter: Get-Alias
DEBUG: CommandDiscovery Information: 0 : Cmdlet found: Get-Alias
Microsoft.PowerShell.Commands.GetAliasCommand,
Microsoft.PowerShell.Commands.Utility, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35
```

CommandType	Name	Definition
-----	----	-----
Alias	ii	Invoke-Item

As you can see, the `gal` command got resolved to the `Get-Alias` cmdlet. Going further, we see that `Get-Alias` is implemented by the `Microsoft.PowerShell.Commands.GetAliasCommand` .NET class. The `PSHost` parameter instructs `Trace-Command` to log messages to the console host window.

Let's now investigate how variables are bound to cmdlet parameters when invoking a cmdlet. We will use the same command as before, changing the component name to `ParameterBinding`:

```
PS> Trace-Command -Name ParameterBinding -PSHost { gal ii }
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args
[Get-Alias]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args
[Get-Alias]
```

```

DEBUG: ParameterBinding Information: 0 :      BIND arg [ii] to
parameter [Name]
DEBUG: ParameterBinding Information: 0 :      Binding collection
parameter Name: argument type [String], parameter type
[System.String[]], collection type Array, element type
[System.String], no coerceElementType
DEBUG: ParameterBinding Information: 0 :      Creating array with
element type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 :      Argument type String
is not IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 :      Adding scalar element
of type String to array position 0
DEBUG: ParameterBinding Information: 0 :      BIND arg
[System.String[]] to param [Name] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on
cmdlet [Get-Alias]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING ProcessRecord
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing

```

CommandType	Name	Definition
-----	----	-----
Alias	ii	Invoke-Item

We get a lot of information again: the `ii` value was interpreted as a positional parameter, and it was interpreted as the `Name` parameter. After checking for missing mandatory parameters, the shell went on to process the pipeline for that cmdlet by calling the `BeginProcessing`, `ProcessRecord`, and `EndProcessing` pipeline execution steps. Those steps are the cmdlet analogs to the `begin`, `process`, and `end` blocks that we need to process pipelines from functions, scripts, and script blocks.

Last, but not least, we will use `Trace-Command` to get information about type casts or conversions. We will need the `TypeConversion` component name. Here is how to get information about implicit conversions that occur when calculating an expression (remember to revert `Set-PSDebug` back to its default setting).

```

PS> Set-PSDebug -Off
DEBUG:      1+ Set-PSDebug -Off
PS> Trace-Command -Name TypeConversion -PSHost { 1 + 0.5 }
1.5

```

Hmm—we got nothing! Remember the part about the `DivideByZeroException` error that we cannot trap? We hit the same problem here: the `1 + 0.5` expression is computed at parse time, and the `Trace-Command` is really given a `1.5` parameter that triggers no type conversions. Let's delay calculation to the actual runtime by moving the values to variables:

```

PS> $a = 1
PS> $b = 0.5
PS> Trace-Command -Name TypeConversion -PSHost { $a + $b }
DEBUG: TypeConversion Information: 0 : Converting "1" to

```

```

"System.Double".
DEBUG: TypeConversion Information: 0 : Original type before
getting BaseObject: "System.Int32".
DEBUG: TypeConversion Information: 0 : Original type after getting
BaseObject: "System.Int32".
DEBUG: TypeConversion Information: 0 : Standard type conversion.
DEBUG: TypeConversion Information: 0 : Converting integer to
System.Enum.
DEBUG: TypeConversion Information: 0 : Type conversion from
string.
DEBUG: TypeConversion Information: 0 : Custom type conversion.
DEBUG: TypeConversion Information: 0 : Parse type conversion.
DEBUG: TypeConversion Information: 0 : Constructor type
conversion.
DEBUG: TypeConversion Information: 0 : Cast operators type
conversion.
DEBUG: TypeConversion Information: 0 : Looking for
"op_Implicit" cast operator.
DEBUG: TypeConversion Information: 0 : Cast operator for
"op_Implicit" not found.
DEBUG: TypeConversion Information: 0 : Looking for
"op_Explicit" cast operator.
DEBUG: TypeConversion Information: 0 : Cast operator
for "op_Explicit" not found.
DEBUG: TypeConversion Information: 0 : Could not find
cast operator.
DEBUG: TypeConversion Information: 0 : Cast operators type
conversion.
DEBUG: TypeConversion Information: 0 : Looking for
"op_Implicit" cast operator.
DEBUG: TypeConversion Information: 0 : Cast
operator for "op_Implicit" not found.
DEBUG: TypeConversion Information: 0 : Looking for
"op_Explicit" cast operator.
DEBUG: TypeConversion Information: 0 : Cast
operator for "op_Explicit" not found.
DEBUG: TypeConversion Information: 0 : Could
not find cast operator.
DEBUG: TypeConversion Information: 0 : Conversion
using IConvertible succeeded.
DEBUG: TypeConversion Information: 0 : Converting
"0.5" to "System.Double".
DEBUG: TypeConversion Information: 0 : Result type
is assignable from value to convert's type
1.5

```

That is much different: we get a full log of the strategies that PowerShell tries in order to convert the 1 integer to a double. It tries to find implicit and explicit cast operators and, failing in that, defaults to converting the value through the .NET `IConvertible` interface that both integer and double numbers support. To get more information about how the shell converts between types, refer to Chapter 1.

Command tracing is a complex topic and can quickly drown you with useless information if you do not use it carefully. Tracing is the heavy-duty tool for script debugging, as it can literally tell us everything about how the shell works. I personally first use some print debugging and resort to tracing only when I hit a wicked problem that I cannot figure out in any other way.

Summary

The topics of error handling and script debugging are intricately connected, and hopefully, this chapter showed exactly that. We can beef up our error handling logic so that we create clear and self-diagnosing code, which, in turn, will require less debugging in order to make it do what we really want. The opposite connection is also true—we can use purely error-handling related techniques, such as error traps, to log error information that helps us in debugging our code.

Adding good error handling capabilities to code is both an important habit and a valuable skill that anyone working with code can benefit from. Given that, most of the time we spend writing code is spent not in typing or planning but in debugging code that we have just written, employing all techniques to reduce debugging time makes sense. Debugging time grows proportionately to script size, and we will use the techniques from this chapter in creating more complex scripts later in this book.



Signing Scripts

Imagine that one of your duties at work involves performing a tedious manual task every day. One evening, just before you leave work, you get an e-mail from a coworker that reads, “Hey, I created this nice PowerShell script that you can run, and it will automatically do the XYZ task for you.”

You are thrilled! You extract the attachment and immediately run the script. Unfortunately, it just spews out an incomprehensible error message and stops. You sigh, send a quick “Your script does not work” message to your coworker and go home.

The next morning, you notice your computer is acting very strange. You dig deeper and find a key logger installed on your machine capturing passwords and sending them who knows where. Adding to that, your firewall has mysteriously opened several important ports to the outside world, and your company president storms into your office demanding an explanation for that “Own3d by Evil Borat” red banner on your company web site. Your network security has been severely compromised. Ouch! You should have read through that script file before running it!

Well, the PowerShell designers have seen this coming. Such scenarios have long been plaguing administrators using other scripting technologies like Windows Scripting Host, and fortunately, you do not have to tremble with paranoia as you read every script every time before running it. PowerShell has a mechanism for signing scripts, so that you know where they come from and that they have not been tampered with. We will explore signed scripts in this chapter, so that none of you, dear readers, loses your job over such a mistake.

How Code Signing Works

Signing code and making 100 percent sure that a program will not harm us is a complex operation with some pretty serious theory behind it. When dealing with signed executable code, we have two problems: the integrity of the code and its origin. We need to know that a script has not been tampered with. For example, we have to make sure somebody has not intercepted Mike’s script and inserted a `del C:*.*` command or something even worse. In addition, we need to know for certain that the code originated with the same person that it claims to be from. In other words, was that script claiming to come from Mike really written by Mike?

Guaranteeing Script Integrity

To verify that we are working with a script in its original form without any additions or deletions, we can attach extra information about the script when we send it to others. Computer scientists have come up with functions that can compute a checksum for a stream of data; those can help us here. Some of the most popular algorithms that can calculate a checksum are MD5 and SHA-1. Those functions, sometimes called hash functions, go over all the data and calculate a number that is highly unlikely to be the same for another stream of data. Changing even a single bit in the original data will yield a completely different checksum value. So, in the case of our friend Mike sending us a script, we really need him to calculate a checksum for the script and include that in his message. We will calculate the checksum for the script using the same algorithm once we get the message. If the two checksum values are the same, then the script has not been modified.

The hash function scheme has one flaw. A malicious person can intercept the message, modify the script, recalculate the checksum, and replace the original checksum with the new one. We need a way to guarantee that the checksum has not been modified too. We can encrypt the checksum, but we are left with the problem of transmitting the encryption key. It is pointless to include the key in the message—an attacker will simply modify the script and reencrypt the new checksum using the key. This problem cannot be solved using one-key, or symmetric, encryption. We need to encrypt the key using an asymmetric algorithm. Asymmetric encryption algorithms use two keys: one to encrypt the data, another to decrypt it. The encryption key, also called the private key, is never transmitted. The decryption key, or the public key, can be freely transmitted to everyone. So here is our complete solution: Mike calculates a checksum for his script, encrypts it with his private key, and sends us a message containing the script, the encrypted checksum, and his public key. We can then use the public key to decrypt the checksum, calculate another checksum, and compare the two values. If they are equal, then we are safe running the script.

Certifying the Origin of a Script

The scheme to certify script integrity looks perfect—until we realize that a malicious person can again intercept the message, modify the script, update the checksum, encrypt it with her private key, and then replace the included public key with her public key. To protect ourselves from that type of attack, we need a way to tell that the public key we got is Mike's key and not somebody else's. We cannot do this all by ourselves. We need a third-party entity that we trust to keep record of everybody's public keys and confirm for us that a key really belongs to the person or organization we believe it to. Today, this task is performed by institutions known as certification authorities (CAs). The most popular ones are companies like VeriSign and Thawte.

Certification authorities issue all public/private key pairs and can tell if a public key really belongs to a given person or organization. When we purchase a digital identity from a CA, essentially, we get a public/private key pair. We receive our keys in a private key file and a certificate

file that contains the public key and additional information such as the name of the certificate holder, the expiry date, and so on. Certificates can be distributed to anyone, and anyone can verify if they belong to the right person. How do we verify that? Along with the public key and the information about the certificate holder, certificates carry a signature—a checksum encrypted with the certificate authority’s private key. Anyone can obtain the public key for the CA; in fact, the keys for the most popular authorities are already available with any Windows installation. Using a public key, anyone can verify a certificate’s integrity.

Back to our problem: Mike buys a digital identity from a certificate authority, and we start again. He calculates a checksum for the script and uses his private key to encrypt it. He then sends the script, the encrypted checksum, and his certificate (containing his public key). Upon receiving his message, we use the certificate authority’s public key to verify if Mike’s certificate is authentic. If it is, we extract Mike’s public key from the certificate and use it to decrypt the script checksum. Now, we can calculate the checksum ourselves and compare our value to the decrypted one. If the two are equal, we are safe running the script. This approach cannot be broken, because an attacker will not have the CA private key and he cannot issue a fake certificate. CAs take all the extra care and responsibility for preventing anyone from getting hold of their private keys.

Note The mechanism for exchanging keys and certifying the origin of the script we just described is commonly referred as public key infrastructure (PKI). It is widely used to establish trust on the Internet and has been used by people to verify different signatures and establish and verify trust to other persons or entities. The full PKI standard includes a lot more than binding a public key to a user identity. For example, it includes mechanisms for revoking certificates that have expired or have been compromised. In this chapter, I will cover signatures and key exchanges only.

Managing Certificates

Digital certificates are all around us. Many activities, not limited to secure sites we trust or signed programs we run, require that we exchange keys and keep track of certificates. Once we trust a software publisher, we do not need to confirm that we trust her products every time we use them. All trusted certificates get stored in the Windows certificate store, and programs like PowerShell get certificate information from that store. The easiest way to manage certificates is through the Microsoft management console tool and its Certificates snap-in. To start that, type **mmc** in your Windows Vista Start Menu search box or in the Run dialog box, if you are running a different version of Windows. Once the management console tool starts, choose Add/Remove Snap-in from the File menu. You should see a dialog box similar to the one shown in Figure 10-1. Pick the Certificates snap-in, as shown in Figure 10-1.

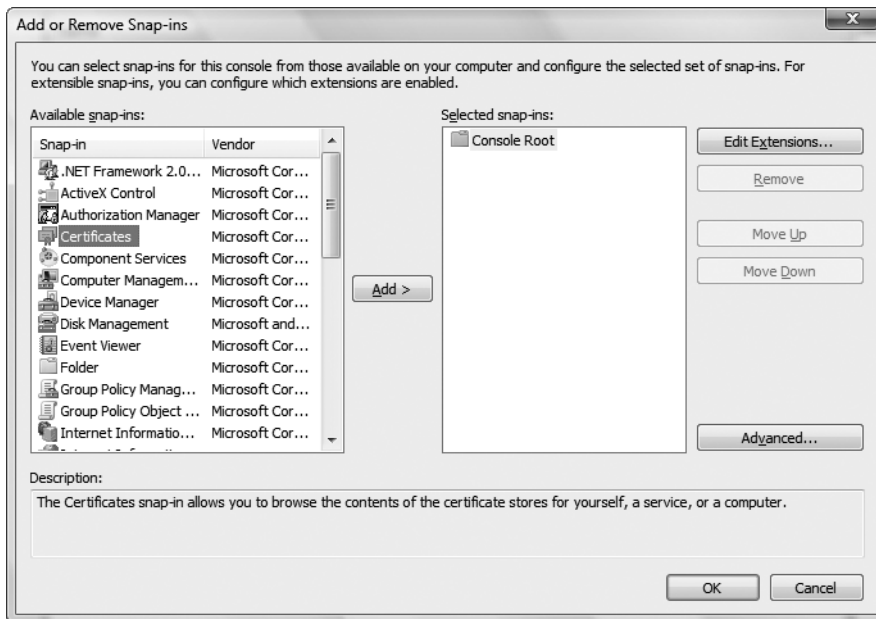


Figure 10-1. Adding the Certificates snap-in

Click the Add button, and click OK. You will be presented with a dialog similar to the one shown in Figure 10-2 that asks what the snap-in will manage.

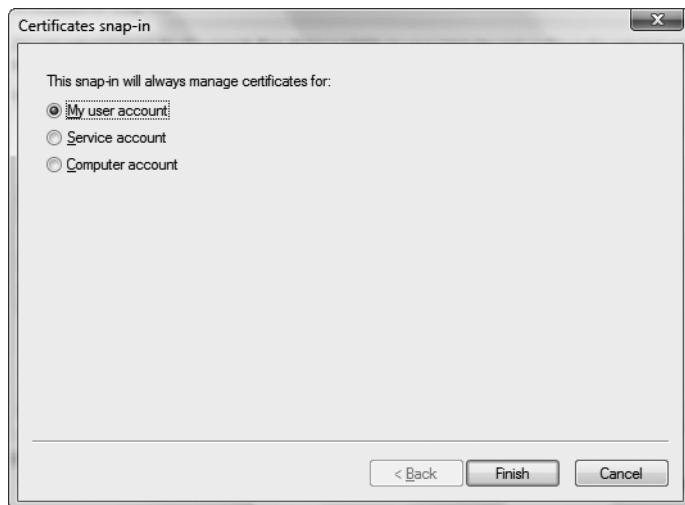


Figure 10-2. Choosing what certificate store to manage

Select that you want to manage certificates for your user account, and click the Finish button. We will work with certificates stored in the current user certificate store in this chapter. If you need to run signed PowerShell scripts under a service account or under all accounts on the

machine, add another instance of the Certificates snap-in, this time choosing to manage certificates for a service account or a computer. Figure 10-3 shows how the management console looks with two snap-ins installed: one for the current user and another for the computer.

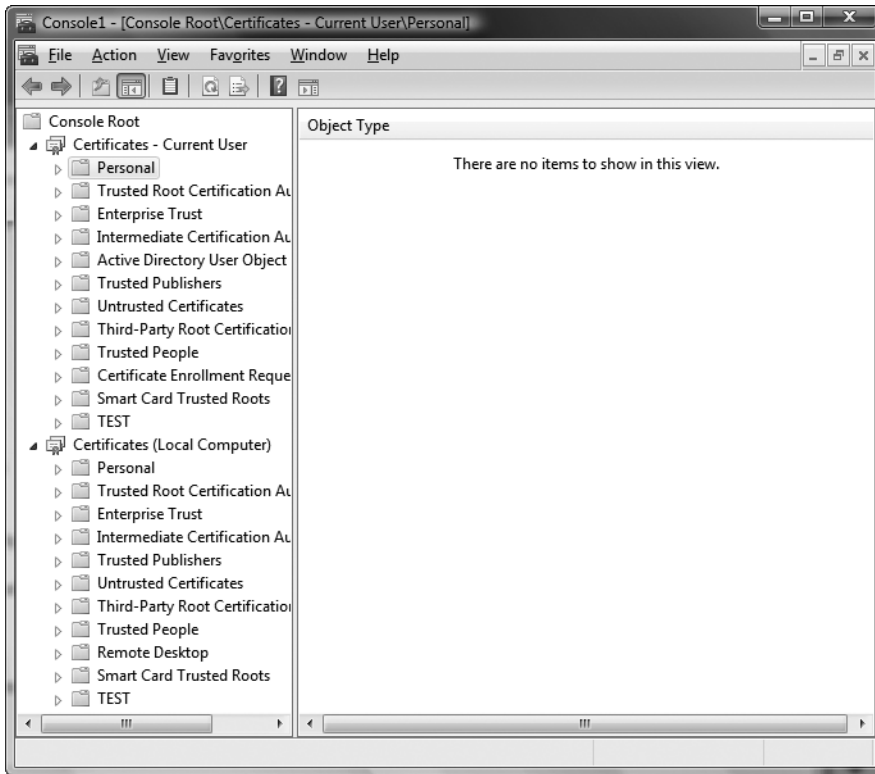


Figure 10-3. *Managing user and system certificates*

As you can see, we have a number of certificate stores that look like folders, for both the current user account and the entire system.

We can access certificates from within PowerShell too. By default, the certificate provider registers a `cert:` drive that we can navigate as if we were navigating the file system. Here is how we can use that to enumerate the certificate stores for the current user:

```
PS> dir cert:\CurrentUser
```

```
Name : SmartCardRoot
```

```
Name : UserDS
```

```
Name : AuthRoot
```

```
Name : CA
```

Name : Trust

Name : Disallowed

Name : My

Name : Root

Name : TrustedPeople

Name : TrustedPublisher

Note that we are getting the same results that we do from the management console tool. The only difference is that the management console snap-in displays friendly names like Personal or Trusted Root Certification Authorities, while the PowerShell provider uses the store short names like My and AuthRoot.

Creating a Self-Signed Certificate

Getting a code-signing certificate provides us the best level of security, but that is not always the most practical thing to do. Requesting a certificate from a certification authority takes some time, as the CA has to run background checks and verify your identity before issuing a certificate—not to mention that a certificate is not cheap. Using the code signing tools and the infrastructure provided by PowerShell, we can achieve a good balance of security and practicality by issuing our own self-signed certificate. We can create a self-signed certificate and use it to sign scripts much in the same way as we would use a certificate issued by a third-party certification authority. Self-signed certificates are just like third-party certificates and can be used to verify a signature. The only problem with them is that they do not carry the same degree of trust, as everyone can create such a certificate, and not everyone has the resources to protect his private key. Most of the time, this limited protection is not a significant problem when distributing scripts in a single network—you know the certificate issuer and know that he knows how to protect the certificate private key from unauthorized access.

Creating a Certification Authority Certificate

To issue a code signing certificate, we have to create a certification authority certificate first. The CA certificate has to be imported to the local certificate store, so that it can be used to verify the script signature. We will use the `makecert.exe` tool to create the certificates. `makecert.exe` is a part of the Microsoft .NET 2.0 Framework SDK which is freely available for download. If you have the SDK installed, `makecert.exe` may already be in your PATH, and you should be able to invoke it by just typing its name. If the Framework SDK executables' folder is not in your PATH, you can add it yourself: just search for `makecert.exe` in your Program Files folder, and add the folder that contains the executable to the system PATH. So, pick a directory that will store the generated certificates and keys, and navigate there. Then, call `makecert.exe` using the following parameters:

```
PS> cd C:\PowerShell\certs
```

```
PS> makecert -n "CN=Pro Windows PowerShell Certification Authority" `
```

```
-a sha1 -eku 1.3.6.1.5.5.7.3.3 -r -ss Root -sr CurrentUser `
-sv ProPowerShell_CA.pvk ProPowerShell_CA.cer
```

Succeeded

Before seeing the “Succeeded” message, you will be presented with several dialog windows that request your input. First, you have to pick a password for your private key file; Figure 10-4 shows how key password dialog.



Figure 10-4. *Setting a private key password*

The rule when dealing with keys is that all private keys, stored in files, are password-protected. This adds an extra layer of security and prevents others from signing code even if they manage to steal your private key. After setting a private key password, you will be asked to confirm it once again. Figure 10-5 shows the password confirmation dialog you can expect.



Figure 10-5. *Confirming the private key password*

After confirming the password, the certification authority certificate will automatically be added to the current user certificate store. Windows will display a warning dialog that reminds you that trusting this certificate means that you automatically trust all certificates issued by that entity. Figure 10-6 shows the rather scary looking dialog.

Confirming that you want to add the certificate is the last step. You can find the certificate imported under the Trusted Root Certification Authorities folder, and you can delete it later if you change your mind about trusting that authority. Figure 10-7 shows what your trusted root certification authorities list looks like after adding our certificate there.



Figure 10-6. Adding the certification authority certificate to the certificate store

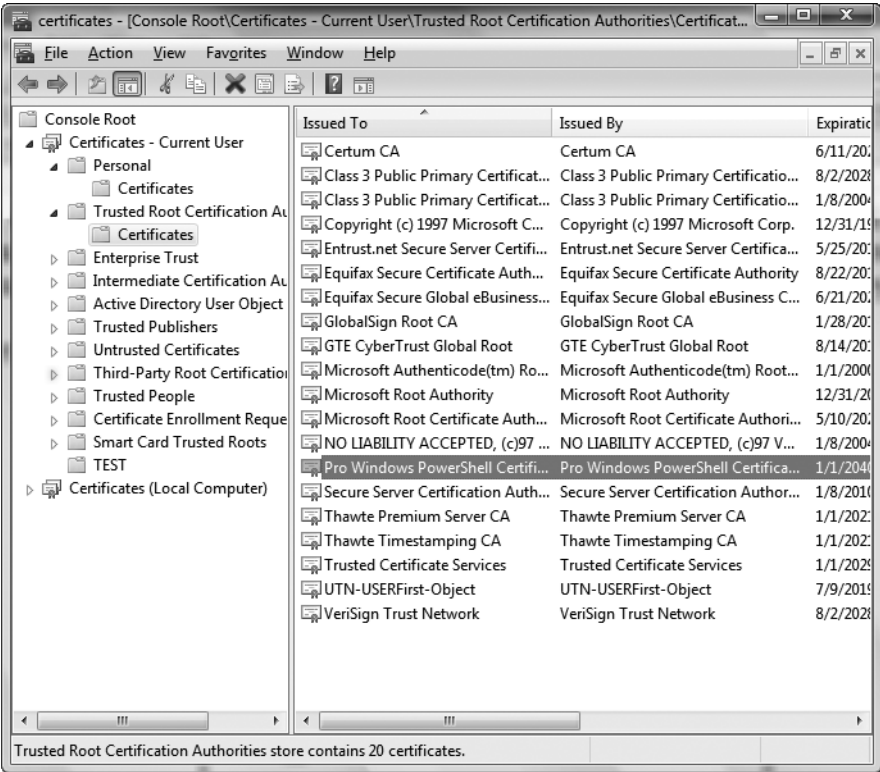


Figure 10-7. Viewing the trusted certification authorities

Double-clicking the certificate will display additional information. You will see something that looks like Figure 10-8.

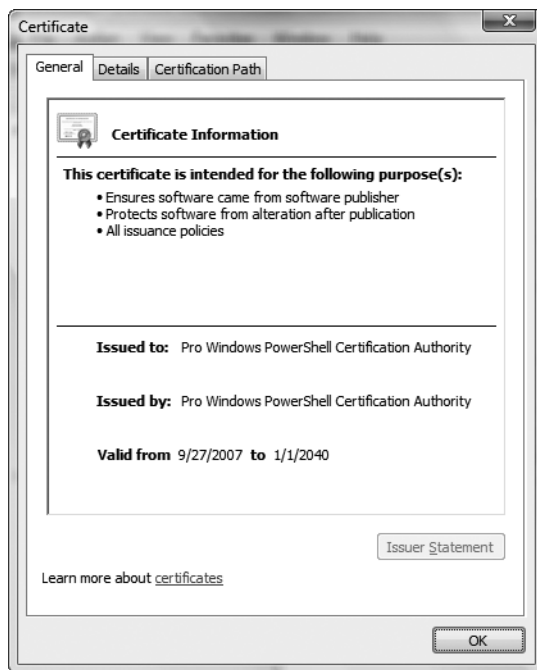


Figure 10-8. Viewing the certification authority certificate details

As you can see from the screenshot, the certificate is valid until January 1, 2040. That is quite a long time!

Now, let me take the time to explain the cryptic options we passed to `makecert.exe` at the command line:

- `-n`: This is the name of the certificate holder. It can be any name that conforms to the X.500 standard. Standards aside, the simplest way to provide a name is to use the `CN=<your name>` syntax.
- `-a`: This is the signature algorithm. We are using the SHA-1 algorithm by specifying the `sha1` option. Another alternative is the MD5 algorithm. Switching to that requires that we use `-a md5`. The signature algorithm is the algorithm used to compute the checksum that guarantees the certificate file integrity.
- `-eku`: This option specifies the enhanced key usage object identifier. We pass `1.3.6.1.5.5.7.3.3`, which is a known standardized constant. To spare you the trouble of reading the X.500 standard, the constant marks the certificate as one that will be used for code signing. `makecert.exe` can create all types of certificates; for example, passing `1.3.6.1.5.5.7.3.1` will create a server authentication certificate that can be used for SSL communication.

- `-r`: This issues a self-signed certificate and tells `makecert.exe` that it should not use a CA's private key to issue this certificate.
- `-ss`: This is the name of the certificate store. This is the location of the certificate store that the newly created certificate will be imported into. We pass `Root`, because we want the certificate to be imported as a root certification authority certificate.
- `-sr`: This is the certificate store location. There are two valid options here: `CurrentUser` and `LocalMachine`. The former adds the certificate for the current user; the latter, for all users on the machine. Note that you need administrator privileges if you wish to add the certificate to the `LocalMachine` store. For Windows Vista, that requires elevated privileges, and you have to start your PowerShell session as an administrator.
- `-sv`: This is the path to the private key file. `makecert.exe` will create this file and store the certificate private key inside. The file will later be used to sign newly issued certificates. In our case, this is `ProPowerShell_CA.pvk`—the file that gets password protected.
- `ProPowerShell_CA.cer`: This is the name of the file that will contain the certificate. As I mentioned before, it contains the public key and additional data about the certificate holder.

Issuing a Code-Signing Certificate

Having a certificate authority now allows us to issue a code-signing certificate. Again, we will use `makecert.exe`, but this time, we will point it to the certificate authority's private key and public certificate. Here is the command:

```
PS> makecert -n "CN=Pro Windows PowerShell Script Publisher" `
    -ss MY -a sha1 -eku 1.3.6.1.5.5.7.3.3 -pe `
    -iv ProPowerShell_CA.pvk -ic ProPowerShell_CA.cer
```

Succeeded

Again, before we get to the “Succeeded” message, we will be prompted for the certificate authority private key password. The prompt dialog should look like the one shown in Figure 10-9.



Figure 10-9. *Entering the certification authority private key password*

The `makecert.exe` tool does not have the most descriptive options in the world, so let's go through each of them:

- `-n`: This is the certificate holder name. Since this is our script publisher certificate, we type **CN=Pro Windows PowerShell Script Publisher**. Use a value that will help you easily distinguish between the certification authority and the script publisher certificate.
- `-ss`: This is the name of the certificate store that will contain our certificate. This time we type **MY**, because we want the certificate in our personal certificate store. This is not a certification authority certificate and has to be placed in a different location.
- `-a`: This is the checksum algorithm. Again we're using SHA-1.
- `-eku`: This is the enhanced key usage parameter. Just as with the certification authority certificate, we pass **1.3.6.1.5.5.7.3.3** to indicate that the certificate will be used to sign executable code.
- `-pe`: This declares the certificate as exportable and tells `makecert.exe` to store the private key inside the certificate store. This makes it possible for us to export the certificate together with its associated private key. You will see how to do that later in this chapter.
- `-iv ProPowerShell_CA.pvk`: This is the certification authority private key file. The private key is used to sign the certificate so that its authenticity can later be verified.
- `-ic ProPowerShell_CA.cer`: This is the certification authority certificate. It is used to obtain additional information about the certification authority that is included in our certificate.

After creating our certificate, we will get no new files in the current folder; the certificate can be found in the certificate store. Figure 10-10 shows it stored inside the Personal store.

Double-clicking the certificate displays additional information about it. Figure 10-11 shows how that looks for our certificate.

There are several things to note here:

- The certificate purposes are to ensure that software came from a specific software publisher and that it has not been altered after publication.
- The certificate was issued to Pro Windows PowerShell Script Publisher; this is the certificate holder name that we provided to `makecert.exe`.
- The certificate was issued by Pro Windows PowerShell Certification Authority, our personal certification authority.
- The key icon indicates that this certificate has a private key associated with it in the certificate store. This is the effect of the `-pe` parameter that we used when generating the certificate.

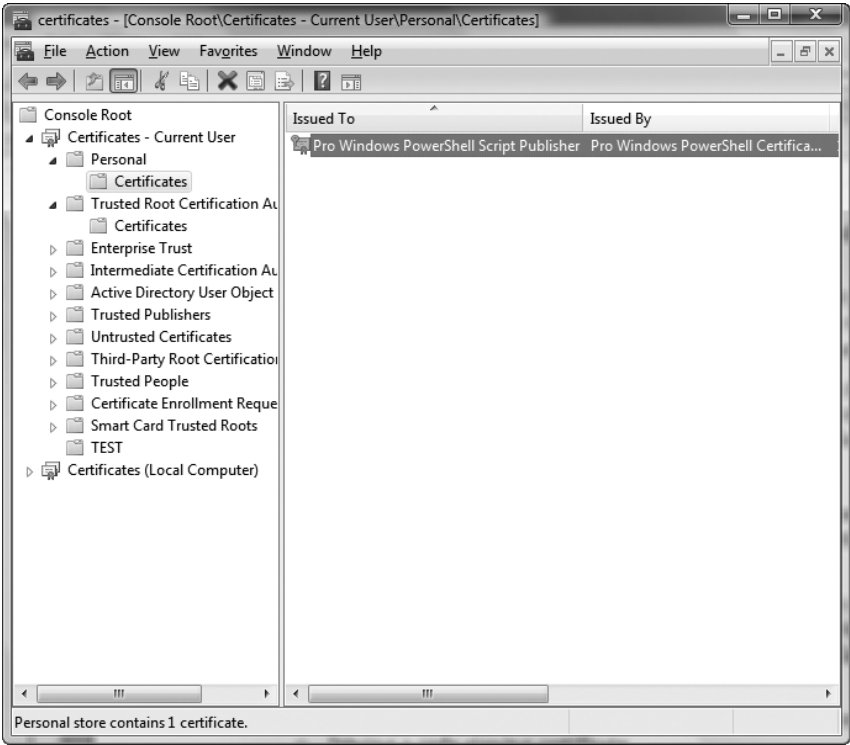


Figure 10-10. *Our script-signing certificate stored in the Personal store*



Figure 10-11. *Displaying information about code-signing certificate*

People often need to have a separate copy of their certificates for various purposes, like backup. We can get a normal file for our certificate by exporting it from the store. To do that, we have to right-click the certificate and select Export from the All Tasks menu. We are presented with the certificate export wizard. We should choose to export the private key together with the certificate, as shown in Figure 10-12.



Figure 10-12. *Exporting a certificate along with its corresponding private key*

Next, we have to choose the file format. Figure 10-13 shows the wizard page that does that. The only available file format is a Personal Information Exchange file, which has the PFX file extension. PFX files are files with a standardized publicly supported format that contain both a certificate and the private key associated with that certificate.

We will leave the rest of the options with their default values.

Since we are exporting the private key with this certificate, we are presented with a password configuration dialog. Remember that all files containing private keys are password-protected. Figure 10-14 shows the password configuration wizard page.

We still need to select a name and location for the certificate file and save it. I have named mine ScriptSign.pfx and stored it in the same folder that contains the certification authority key and certificate. That certificate file can now be imported in any certificate store on any machine. In addition, we can use it directly from PowerShell to sign scripts.

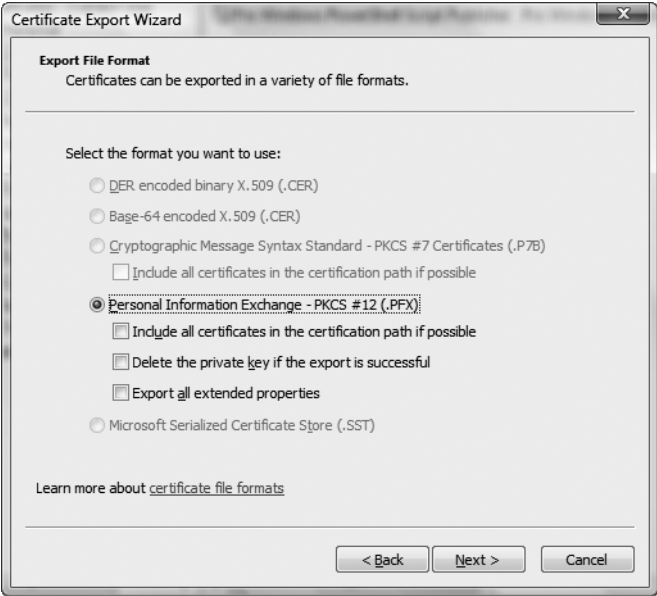


Figure 10-13. *Selecting the certificate export file format*

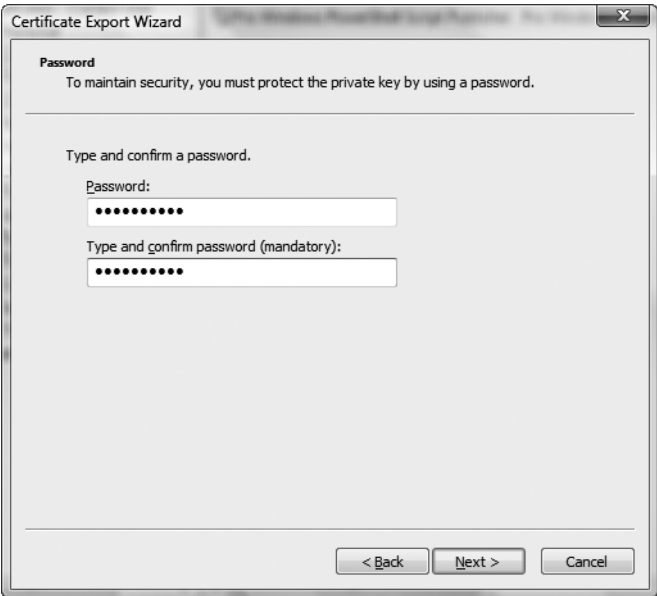


Figure 10-14. *Setting a password for the exported certificate file*

Signing Scripts

Before we sign a script, let's make sure that we require that running a script requires that it has been signed beforehand. To enforce that, we have to switch our current script execution policy to `AllSigned`. Only users with administrator privileges can do that on a machine, and on Windows Vista, those users need to run the shell with elevated privileges. So, start a shell session as an administrator, and call the `Set-ExecutionPolicy` cmdlet:

```
PS C:\> Set-ExecutionPolicy AllSigned
PS C:\> Get-ExecutionPolicy
AllSigned
```

Let's now create a simple script that returns all certificate files in the current folder and try to run it. Here is what happens:

```
PS> Set-Content Get-CertificateFiles.ps1 "dir *.cer"
PS> .\Get-CertificateFiles.ps1
File C:\PowerShell\certs\Get-CertificateFiles.ps1 cannot be loaded. The file C:\PowerShell\certs\Get-CertificateFiles.ps1 is not digitally signed. The script will not execute on the system. Please see "get-help about_signing" for more details..
At line:1 char:19
+ .\Get-CertificateFiles.ps1 <<<<
```

Since we are running under the `AllSigned` script execution policy, we are not allowed to run unsigned scripts. Let's sign it! We need to get hold of our signing certificate and pass it, along with the script path, to the `Set-AuthenticodeSignature` cmdlet.

Let's get the certificate from the certificate store and use it to sign the script:

```
PS> $certStore = dir cert:\CurrentUser\My
PS> $certStore
```

```
Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\My
```

Thumbprint	Subject
-----	-----
E5F1C5F21B5E276374626A9C4F68E39DBB0BB062	CN=Pro Windows PowerShell Scri...

```
PS> Set-AuthenticodeSignature Get-CertificateFiles.ps1 -cert $certStore
```

```
Directory: C:\PowerShell\certs
```

SignerCertificate	Status	Path
-----	-----	-----
E5F1C5F21B5E276374626A9C4F68E39DBB0BB062	Valid	Get-Certifica...

Note how we get the certificate object. My personal store contains only one certificate. If it contained more, I would have had to use an indexer like `(dir cert:\CurrentUser\My)[0]` to get to the correct certificate. As you can see in the preceding example, `Set-AuthenticodeSignature` returns the script signature. You can get a script's signature at any time by calling `Get-AuthenticodeSignature`:

```
PS> Get-AuthenticodeSignature Get-CertificateFiles.ps1
```

```
Directory: C:\PowerShell\certs
```

SignerCertificate	Status	Path
-----	-----	----
E5F1C5F21B5E276374626A9C4F68E39DBB0BB062	Valid	Get-Certifica...

If the Status property is Valid, the script is safe for execution. We can now run it:

```
PS> .\Get-CertificateFiles.ps1
```

```
Do you want to run software from this untrusted publisher?
File C:\PowerShell\certs\Get-CertificateFiles.ps1 is published by CN=Pro
Windows PowerShell Script Publisher and is not trusted on your system. Only
run scripts from trusted publishers.
[V] Never run [D] Do not run [R] Run once [A] Always run [?] Help
(default is "D"):A
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\PowerShell\certs
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	9/27/2007 5:07 PM	632	ProPowerShell_CA.cer

This is the first time that we have run a script from that publisher. PowerShell notices that and asks if we really trust that guy. It displays information about the publisher, so that we can make an intelligent decision. We decide to always run code from Pro Windows PowerShell Script Publisher and choose A, so that we do not get that question anymore. The script then executes.

What does “trusting a publisher” really mean? When you trust a script publisher, PowerShell extracts the public certificate from the signed script and stores it in your Trusted Publishers certificate store (see Figure 10-15).

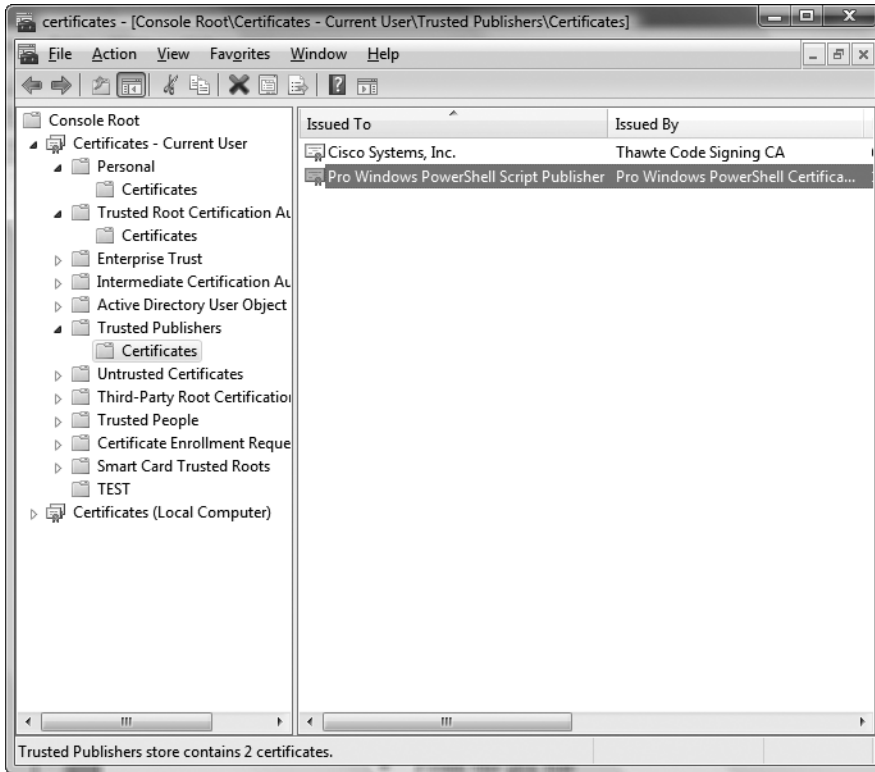


Figure 10-15. Our publisher certificate added to the Trusted Publishers store

Deleting the certificate from the Trusted Publishers store will present you with the “Do you trust this guy?” question the next time you try to run that script.

We can get the publisher certificate from the PFX file we previously exported, and that has both advantages and disadvantages to getting the certificate from the certificate store. To extract the certificate from the PFX file, we have to use the `Get-PfxCertificate` cmdlet:

```
PS> $certFile = Get-PfxCertificate ScriptSign.pfx
Enter password: *****
```

Note that we had to enter the private key password to get the certificate. This feels a bit more secure than the certificate store approach, because getting a certificate from the store did not ask for a password. The certificate store advantage is that it is a far more secure place to store certificates and does not require you to fiddle with file system permissions. Anyway, once you get the certificate, everything about signing a script looks the same:

```
PS> Set-AuthenticodeSignature Get-CertificateFiles.ps1 -cert $certFile
```

Directory: C:\PowerShell\certs

SignerCertificate	Status	Path
-----	-----	----
E5F1C5F21B5E276374626A9C4F68E39DBB0BB062	Valid	Get-Certifica...

```
PS> Get-AuthenticodeSignature Get-CertificateFiles.ps1
```

Directory: C:\PowerShell\certs

SignerCertificate	Status	Path
-----	-----	----
E5F1C5F21B5E276374626A9C4F68E39DBB0BB062	Valid	Get-Certifica...

Here is what the script looks like after signing:

```
PS> type Get-CertificateFiles.ps1
dir *.cer
```

```
# SIG # Begin signature block
# MIIEdAYJKoZIhvcNAQcCoIIeZTCCBGECAPQExCzAJBgUrDgMCGGUAMGkGCisGAQQB
...
# HbdgCU3A58lcpdrhYc2Zzt5n68nAuT01EKaWwKPIqUYTmBaRF2/A==
# SIG # End signature block
```

As you can see, the signature is embedded in a comment section. Let's now verify how secure signing is by changing the script. Let's pretend that a malicious user gets the script and inserts a `del C:*.*` command at the top:

```
del C:\*.*
dir *.cer

# SIG # Begin signature block
# ...
# SIG # End signature block
```

We suspect nothing and run the script:

```
PS> .\Get-CertificateFiles.ps1
File C:\PowerShell\certs\Get-CertificateFiles.ps1 cannot be loaded. The con
tents of file C:\PowerShell\certs\Get-CertificateFiles.ps1 may have been ta
mpered because the hash of the file does not match the hash stored in the d
igital signature. The script will not execute on the system. Please see "ge
t-help about_signing" for more details..
At line:1 char:26
+ .\Get-CertificateFiles.ps1 <<<<
```

PowerShell rightfully complains that the content of the script has changed since the time of its signing. It will protect us by refusing to execute the code inside. Let's now manually verify the signature:

```
PS> Get-AuthenticodeSignature Get-CertificateFiles.ps1
```

```
Directory: C:\PowerShell\certs
```

SignerCertificate	Status	Path
-----	-----	----
E5F1C5F21B5E276374626A9C4F68E39DBB0BB062	HashMismatch	Get-Certifica...

The Status property has a HashMismatch value. That means that the calculated checksum is different than the one embedded with the signature. You can feel safe now! Nobody can trick you into running code that you did not intend to.

Running Scripts on Other Machines

One of the limitations of signing scripts using your own self-signed certification authority certificate is that it is very hard to verify that signature on another machine. Had we been using a VeriSign-issued certificate to sign our scripts, we would have had no problem—the VeriSign root certificate is installed on every Windows machine. To achieve the same, we have to install our certification authority certificate on all machines that have to run our scripts. Here is what happens if we run a signed script on a machine that does not have our certification authority's certificate installed:

```
PS> .\Get-CertificateFiles.ps1
File C:\PowerShell\certs\Get-CertificateFiles.ps1 cannot be loaded. A certi
ficate chain processed, but terminated in a root certificate which is not t
rusted by the trust provider.
At line:1 char:26
+ .\Get-CertificateFiles.ps1 <<<<
```

PowerShell tries to go find the issuer certificate in the Trusted Root Authorities store and fails. That is why we need to export the certificate from the source machine and import it on every destination machine that we need to support.

To export our CA certificate, we have to right-click it and choose Export from the All Tasks context menu option. The second page of the export wizard will ask for the certificate file type. Go with the default DER file format, as shown on Figure 10-16.

That is all. Save the certificate to a CER file, and transfer it to the target machine. Note that you are not asked to provide a password this time. That is because this certificate contains a public key only. It will not be used to sign files, just to verify previously signed ones. And, of course, your certification authority private key is stored only inside that ProPowerShell_CA.pvk file that makecert.exe created the first time. This is the file you have to guard most carefully.



Figure 10-16. *Selecting an export certificate file format*

Once you get to the target machine, open another management console, and add the Certificates snap-in. Navigate to the Trusted Root Certification Authorities certificate store, right-click the Certificates subfolder, and choose Import from the All Tasks context menu. The first step in the import wizard will ask you to navigate to the certificate file. Pick the CER file you exported in the previous operation.

Next, you will be asked to confirm that you really want to import the certificate under the Trusted Root Certification Authorities store; Figure 10-17 shows that.

You will see the same warning that you saw on Figure 10-6 asking you to confirm that you really want to add a new trusted certification authority to your certificate store. Please, double-check the certificate thumbprint before you do so, especially if somebody else has given you the certificate and you have not generated it yourself. Any extra effort spent checking those certificates pays off in hours of good sleep later on!

You will be asked to confirm that you trust the Pro Windows PowerShell Script Publisher entity and whether you really want to run code originating from that person or organization. You can avoid even that by exporting the Trusted Publisher certificate that you saw on Figure 10-15 and importing it in the same store on the target machine. Most often, manually importing the publisher certificate is not worth the hassle, as creating the same entry in the Trusted Publisher store takes a single key stroke.



Figure 10-17. *Importing a new trusted root certification authority certificate*

Summary

Signing code and using checksum and encryption algorithms is a vast topic, and we have just scratched the surface of it. You learned how signing files works, and how code origins and integrity are guaranteed in an inherently insecure medium such as the Internet. You also learned how to exchange keys and certificates and how to manage and configure the authorities whom we trust with verifying our signatures.

Using the tools demonstrated in this chapter, it is perfectly possible and feasible to set up your environment so that all production servers running PowerShell scripts require all scripts to be signed. Signing your scripts and requiring the use of signed scripts really does make your network bulletproof. In addition, signing any type of code is not much different than signing scripts. You are a small step away from requiring all programs running on important machines to be signed by a known publisher or by you, so that you know there will be no unpleasant surprises.



The Shell Environment and Its Configuration

Every day, I continue to be amazed at the extent to which PowerShell can be tweaked and customized. It offers a wide range of configurable options that we can change by modifying variables or functions. This chapter will describe the shell environment, how we can get information about the shell settings, and how to configure the most popular options of the shell. We will work with the initialization scripts, or the so-called user profile scripts, that allow us to save configuration-related code and execute it for all instances of the shell. Finally, I will show how the shell can call our code when building the prompt string or when looking for strings to complete the current command.

Apart from being very customizable, PowerShell has been designed so that it is easy to host its core in your own application. Many products do that to provide all sorts of advanced features, so our script code can run in many different types of PowerShell hosts. You will learn how to set configuration options for a specific shell host and configure it independently. By creating host-specific configurations, we can benefit from a host's advanced features when they are available.

Shell Hosts

The `powershell.exe` program is a console application. That fact, combined with previous experience with different shells, may lead us into thinking about command-line shells as console applications. That may not always be true. For example, the official PowerShell development guidelines explicitly state that no code should assume it is running in a console environment and that it should never ever output text directly to the console. Why is that? One of the design considerations for the shell was that it has to be easy to embed a shell in any application—and that really means any application, not just console ones. Applications that embed PowerShell are called shell hosts, and they can be all sorts of applications—GUI tools, services, and even applications running on a remote machine, in addition to console applications.

Why do people need to host PowerShell in their applications? Reasons vary, but one of the most popular ones is to provide better shell experience than the one that `powershell.exe` provides, including better command-line completion, fully featured debugging, and data visualization. Another popular use for embedding a shell is reusing its services (providers, cmdlets, and so on). The new Microsoft Windows administration model requires that administrator tools do not modify the system state directly. Instead, such tools are supposed to generate PowerShell scripts that do the real work. The benefits of that approach are huge. For starters, those scripts

can be saved and reexecuted later or executed on many machines. You can even host PowerShell in a Microsoft Management Console snap-in and combine the standard administrator GUI interface with the soon-to-be-standard command-line interface.

PowerShell exposes the shell host to scripts so that code running in the shell can communicate with and easily get information about the host and the environment. For example, we can query the host name and ask it to perform user-interface-related services for us. Let's now get the host object and see what it has to offer. We do that through the \$host global variable:

```
PS> $host

Name           : ConsoleHost
Version        : 1.0.0.0
InstanceId      : 042ba062-31ac-42e3-8e49-96730e454a28
UI             : System.Management.Automation.Internal.Host.Internal
                HostUserInterface
CurrentCulture  : en-US
CurrentUICulture : en-US
PrivateData     : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
```

Another way to get to the host object would be to call the Get-Host cmdlet. Both approaches do the same thing with the only difference being that the \$host variable name is shorter to type.

In the preceding code, we get an instance of the default Microsoft.PowerShell.ConsoleHost object that provides the basic hosting services for us. We can query the host object and use the information it provides to find out what host we are running under, for example, the name and the version of the host application. Writing code that does different things under different hosts is generally not advised, but that may sometimes be our only choice: suppose we need to work around a host bug or know if an advanced feature is available before actually using it.

The most important services a host object offers are UI-related operations. While those are not meant to be used directly from scripts and were primarily intended to support cmdlet authors, we can still take advantage of some of them, as you will see later in this chapter. The UI entry point is the \$host.UI property that returns a PSHostUserInterface object. This is the object implementing the essential contract a host has to offer to scripts and cmdlets. You will see that the contract resolves about a command-line interface and involves methods for reading and writing text. Here are the PSHostUserInterface methods:

```
PS> $host.UI | Get-Member -type Method
```

TypeName: System.Management.Automation.Internal.Host.InternalHostUserInterface

Name	MemberType	Definition
-----	-----	-----
Equals	Method	System.Boolean Equals(Object obj)
GetHashCode	Method	System.Int32 GetHashCode()
GetType	Method	System.Type GetType()

get_RawUI	Method	System.Management.Automation.Host...
Prompt	Method	System.Collections.Generic.Dictio...
PromptForChoice	Method	System.Int32 PromptForChoice(Stri...
PromptForCredential	Method	System.Management.Automation.PSCr...
ReadLine	Method	System.String ReadLine()
ReadLineAsSecureString	Method	System.Security.SecureString Read...
ToString	Method	System.String ToString()
Write	Method	System.Void Write(String value), ...
WriteDebugLine	Method	System.Void WriteDebugLine(String...
WriteErrorLine	Method	System.Void WriteErrorLine(String...
WriteLine	Method	System.Void WriteLine(), System.V...
WriteProgress	Method	System.Void WriteProgress(Int64 s...
WriteVerboseLine	Method	System.Void WriteVerboseLine(Stri...
WriteWarningLine	Method	System.Void WriteWarningLine(Stri...

As you can see, the `PSHostUserInterface` methods prompt the user for input, read text input, and write back normal output, diagnostics, and progress messages. We can call those methods directly:

```
PS> $host.UI.WriteLine("Hello")
Hello
```

As previously mentioned, those methods are primarily targeted at cmdlet authors, and calling `$host.UI.WriteLine()` offers no advantages to calling the `Write-Host` cmdlet. As a matter of fact, it is highly likely that if a future version of the host methods changes in a way that is not backward compatible, the `Write-Host` cmdlet will continue to work. In other words, the safer way to write to the host is through the official cmdlet.

The `PSHostUserInterface` object offers high-level functions for working with the shell host. We have a somewhat lower level alternative that we can use to manipulate various interesting aspects of our shell session. We can do that through the raw user interface object. This is an object of the `PSHostRawUserInterface` type that offers access to physical properties of the shell window: the character buffer, the window properties, text foreground and background color, and so on. Here is how we can get the most important properties:

```
PS> $host.UI.RawUI
```

```
ForegroundColor      : Gray
BackgroundColor      : Black
CursorPosition       : 0,299
WindowPosition       : 0,275
CursorSize           : 25
BufferSize           : 71,300
WindowSize           : 71,25
MaxWindowSize        : 71,70
MaxPhysicalWindowSize : 205,70
KeyAvailable         : False
WindowTitle          : PowerShell
```

At the very least, we can use this object to modify the shell session and set our custom color scheme. Assume I do not like the default Windows Vista look of gray text on a black background, and I want a classic Borland IDE look with yellow text on a blue background. Here is how to do it:

```
PS> $host.UI.RawUI.BackgroundColor = "blue"
PS> $host.UI.RawUI.ForegroundColor = "yellow"
PS> Clear-Host
PS>
```

The `Clear-Host` call is needed to force a redraw of the entire console buffer. Figure 11-1 shows how my shell looks after changing the colors.

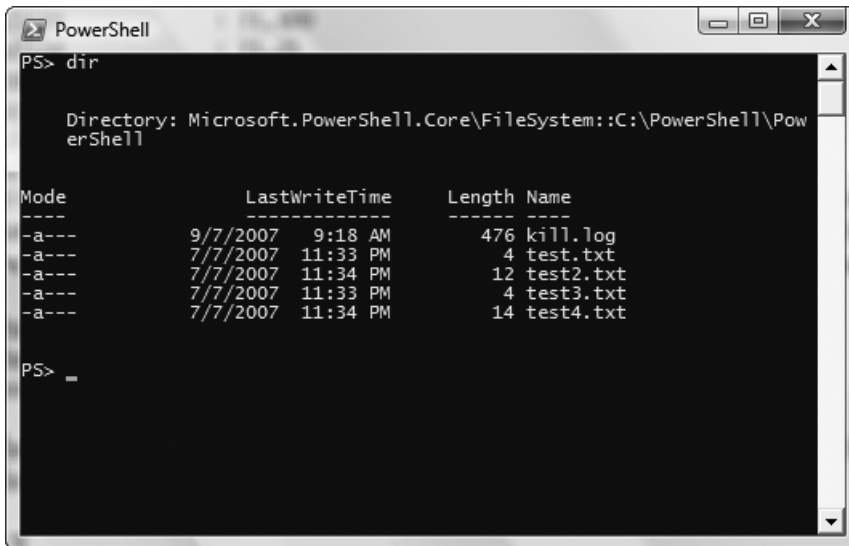


Figure 11-1. A shell window with a modified color scheme

The raw UI offers interesting possibilities for modifying the console window. Here is how we can set the window size and title:

```
PS> $size = $host.UI.RawUI.WindowSize
PS> $size.Width = 50
PS> $size.Height = 20
PS> $host.UI.RawUI.WindowSize = $size
PS> $host.UI.RawUI.WindowTitle = "RawUI Rocks!"
PS>
```

Modifying the window size needs some explanation. It seems that directly setting the `$host.UI.RawUI.WindowSize.Width` and `$host.UI.RawUI.WindowSize.Height` properties does not trigger a user interface update. The host only updates the UI when we set the `WindowSize` property, so we get a reference to the original `WindowSize`, modify it, and assign the same object back to trigger an update. Figure 11-2 shows the end result: a smaller shell window with an odd window title.

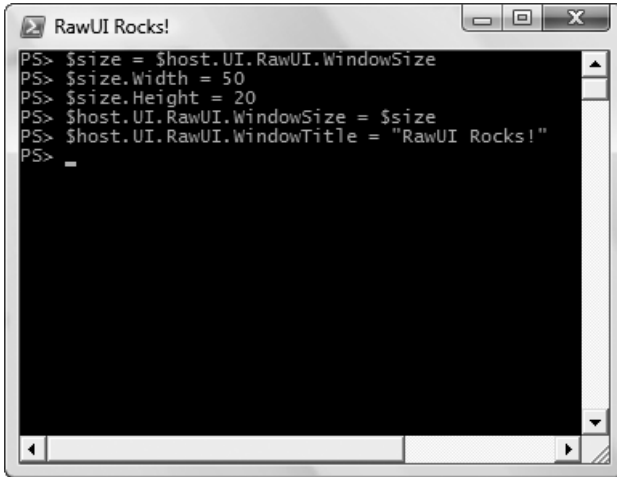


Figure 11-2. *A heavily modified shell window*

User Profile Scripts

Looking at the nice ways to modify a shell session and set up the perfectly styled and configured window, we naturally wish for a way to save those settings and apply them for every console session. We can do that by saving the configuration code in our profile script, a script placed in a location that's known to the shell and that gets executed every time the shell starts. PowerShell offers several options to choose from when saving configuration scripts. According to our needs, we have to pick a different location or a different profile script. The options are explained in the following sections.

Settings for All Users and All Shells

Settings for all shells typically are placed in files named `profile.ps1` that are located inside the Windows PowerShell install folder. To get to that location in PowerShell, use the `$PSHome` variable like this:

```
PS> $PSHome
C:\Windows\System32\WindowsPowerShell\v1.0
```

And here is how to add a setting there:

```
PS> $code = "Write-Host 'Hello, all users, all shells'"
PS> Add-Content $PSHOME\profile.ps1 $code
PS>
```

Note that, because you are modifying a file in the system folder, you have to run the preceding code with administrator credentials and elevated privileges (if running under Windows Vista).

Settings for All Users Using a Specific Shell

To create settings for all users of a specific shell, we again have to place a file in the PowerShell install folder. This time, the file is named according to the convention `<ShellID>_profile.ps1`. In the case of the default shell host, this is `Microsoft.PowerShell_profile.ps1`. Here is how to add a setting that will take effect for all users running the default shell host:

```
PS> $code = "Write-Host 'Hello, all users of Microsoft.PowerShell'"
PS> Add-Content $PSHOME\Microsoft.PowerShell_profile.ps1 $code
PS>
```

Just like I mentioned before, you will need administrator credentials and elevated privileges to run the code.

How do we get the `ShellID` value for an arbitrary shell host? Doing that is as simple as querying the PowerShell `$ShellID` variable:

```
PS> $ShellID
Microsoft.PowerShell
```

Settings for a Specific User and All Shells

Just like for all users, to store settings for a specific user and all shells, we have to create a `profile.ps1` file. Again, the file location is what matters here. The user-specific file has to be placed inside a `WindowsPowerShell` folder that itself resides inside the user `Documents` folder. Under Windows Vista, that file should be placed in `C:\Users\<UserName>\Documents\WindowsPowerShell`. For earlier versions of Windows, like Windows XP, the `WindowsPowerShell` folder will be placed inside the `My Documents` folder. Here is how to add a setting for all shells for the specific user:

```
PS> $code = "Write-Host 'Hello, Hristo, all shells'"
PS> $UserProfileDir = "C:\Users\Hristo\Documents\WindowsPowerShell"
PS> Add-Content $UserProfileDir\profile.ps1 $code
PS>
```

This time, we are modifying a user-specific file, and we do not need administrator privileges.

Settings for a Specific User and a Specific Shell

Code that should run for a specific shell should be placed in a `<ShellID>_profile.ps1`. To make it user specific, we have to save the file inside the `WindowsPowerShell` folder in the user's `Documents` folder that I mentioned in the previous section. For my user account on Windows Vista, the file path is `C:\Users\Hristo\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1`. PowerShell offers a shortcut to get to this path—the `$profile` variable:

```
PS> $profile
C:\Users\Hristo\Documents\WindowsPowerShell\Microsoft.PowerShell_profi
le.ps1
PS>
```

Let's now add some configuration code there:

```
PS> $code = "Write-Host 'Hello, Hristo, Microsoft.PowerShell shell'"
PS> Add-Content $profile $code
PS>
```

The preceding examples use `Add-Content` as the easiest way to add some code to our profile script files. That operation will not have to be called often and does not need to be scripted from within PowerShell. It is best if you use a text editor like `notepad.exe` to open those files and add your code that way.

Now that we have added all sorts of configuration code, my shell spews four startup messages. Figure 11-3 shows the shell window at startup.

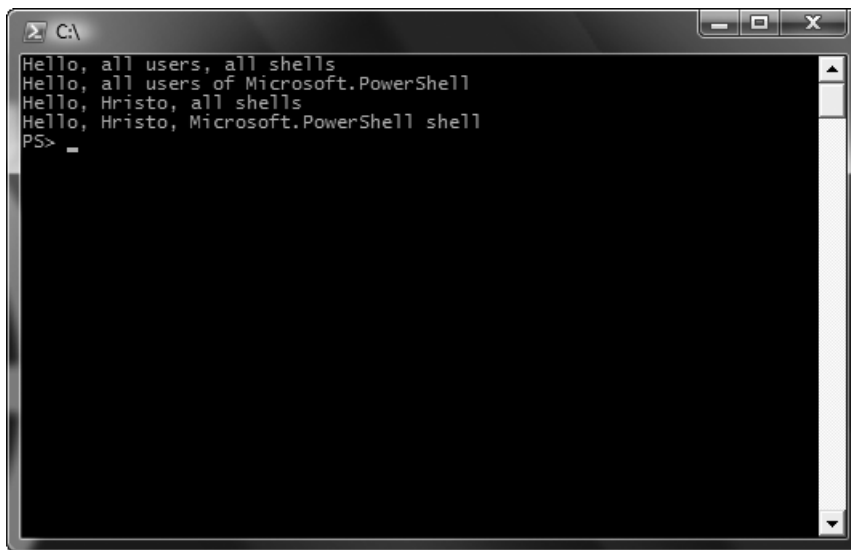


Figure 11-3. *PowerShell executing startup code from various profile scripts*

You might be asking, “What is really the point of having four possible locations for saving your shell profile code?” The all-users profile scripts are useful from a system administration point of view; being able to push common shell settings to all users can be extremely useful for administrators. I really like the idea of being able to have host-specific configuration code, too. I like to have a visual distinction for the various shells that I run, as that minimizes my time looking at error messages caused by me typing stuff in the wrong shell window. I usually color-code different hosts’ text or background, so that my brain knows that a given command goes in the red text console window, for example. PowerShell does that too—the default shell has a blue background when run with elevated privileges on Windows Vista, so you immediately know that commands you run there may have more destructive consequences than usual.

Working with Saved Console Settings

PowerShell can be extended with custom snap-ins. A snap-in is a .NET assembly, usually a DLL file, that contains cmdlets and/or providers. There are a lot of useful extensions on the Internet already, and some of them have gotten quite popular. The process of using a snap-in involves two steps:

1. Register the snap-in. This requires administrator credentials and elevated privileges and is usually done by the snap-in installer.
2. Include the snap-in in the current shell session. Before using cmdlets or providers from a snap-in, users must use the `Add-PSSnapIn` cmdlet to add the snap-in to the current shell session.

As an example for including a snap-in, we can add the `PSEventing` snap-in that contains the cmdlets for the PowerShell Eventing Library (I will discuss the PowerShell Eventing Library in detail in Chapter 22). Here is how to add the snap-in:

```
PS> Add-PSSnapin PSEventing
PS>
```

We can now confirm that the snap-in was really added to our shell session by calling the `Get-PSSnapIn` cmdlet:

```
PS> Get-PSSnapin
```

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

```
Name       : PSEventing
PSVersion  : 1.0
Description : PowerShell Eventing Library 1.0
```

The list starts with the PowerShell built-in snap-ins, and at the end, we see our PSEventing addition. Now comes the tricky part—we have to issue that `Add-PSSnapIn` call every time we start a new PowerShell session. One way to get rid of that nuisance is to add the code to our profile script. Another is to export the shell configuration; PowerShell can load that configuration on startup and restore the included snap-ins.

So, how do we export the shell configuration? We have to use the `Export-Console` cmdlet. It will generate a configuration file that we can use later. Console settings files, by convention, have the `PSC1` file extension. Finally, here is how to export our console settings:

```
PS> Export-Console C:\PowerShell\PSEventing.psc1
PS>
```

Console configuration files are very readable XML files. Here is what our configuration looks like:

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

Now, whenever we want to have a shell that has the PSEventing snap-in included, we have to run PowerShell passing that configuration file as the `-psconsolefile` parameter. Here is how to start PowerShell from within `cmd.exe`:

```
C:\>PowerShell.exe -psconsolefile C:\PowerShell\PSEventing.psc1
Windows PowerShell
Copyright (C) 2006 Microsoft Corporation. All rights reserved.
```

```
PS C:\> Get-PSSnapin PSEventing
```

```
Name       : PSEventing
PSVersion  : 1.0
Description : PowerShell Eventing Library 1.0
```

As you can see, our snap-in has been included, and we can use its cmdlets right away. As a convenience feature, once PowerShell installs on your system, it will associate `.psc1` files with Windows so that double-clicking them will open a shell that uses the respective configuration.

I usually keep shortcuts to several configuration files on my desktop that start the appropriately configured shell when I need it.

Unfortunately, PowerShell saves only included snap-ins in its configuration files. You have to save other configuration data on your own. For example, new aliases that you have configured need to be exported independently using the `Export-Alias` cmdlet and then imported back using `Import-Alias`. Please refer to Chapter 6 for additional information on alias export and import operations.

Changing the Prompt Settings

All shells that I have seen, `cmd.exe` and UNIX shells included, have a mechanism to change the prompt string that gets displayed to the user before a command executes. Those shells allow users to configure the prompt by setting a global or an environment variable. For example, the `cmd.exe` default prompt has the `PG` value, where `$P` makes the shell print the current drive and path and `$G` prints the greater-than, or `>`, sign. The documentation lists several other options, all starting with a dollar sign, that one can use. UNIX shells, such as `bash`, offer a lot more options, but in essence, the approach is the same, including options and literal strings in a global variable. This approach has several disadvantages:

- It is hard to understand and document, as setting the prompt string requires that the user learn a whole new language in order to configure the shell prompt.
- The mechanism is not extensible at all. There is no way to insert a string that will vary according to a custom condition if the shell designers have not thought about it.

The PowerShell designers have elegantly removed those obstacles by choosing to configure the shell prompt through a global function. Users can redefine that function and return whatever prompt string they want. This way, the prompt gets extended with PowerShell's own script language instead of using a custom language. In addition, this mechanism is very extensible, as the return value can depend on anything we can implement with PowerShell.

Let's start our prompts adventure by looking at the default prompt value. We get that by looking at the prompt function definition:

```
PS C:\> (dir Function:\Prompt).Definition
'PS ' + $(Get-Location) + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
PS C:\>
```

As you can see, we get the “PS” string, followed by the current path as returned by the `Get-Location` cmdlet. The prompt ends with the `>` sign and it optionally contains `>>` if we are inside a nested prompt. Nested prompts are special prompts that indicate to the user that she is in the middle of typing a complex command, say a function body, and something more is expected to finish the command (in the function body example, that would be a closing brace).

Let's now try to build a prompt that contains the current user name. To get the current user, we will use the .NET framework API that will return the current Windows identity object. We will wrap that in a function and call it `Get-User`. Here is the code:


```
PS> function Get-User
{
    [System.Security.Principal.WindowsIdentity]::GetCurrent()
}
```

```
PS> Get-User
```

```
AuthenticationType : NTLM
ImpersonationLevel : None
IsAuthenticated    : True
IsGuest            : False
IsSystem           : False
IsAnonymous        : False
Name               : NULL\Hristo
Owner              : S-1-5-21-2505111388-2288116029-638358836-1000
User               : S-1-5-21-2505111388-2288116029-638358836-1000
Groups             : {S-1-5-21-2505111388-2288116029-638358836, , ..
                    .}
Token              : 3176
```

Let's now build our prompt that will contain the user name and the current folder followed by a > symbol and a space:

```
PS> function prompt
{
    $user = (Get-User).Name
    $path = Get-Location
    "[$user] $path> "
}
```

```
[NULL\Hristo] C:\>
```

We can use our user object and configure a prompt similar to the de facto standard for many UNIX shells, one that ends with a pound sign (#) if the user is a computer administrator and a dollar sign (\$) otherwise. To determine if the user is an administrator, we will get the current Windows principal for the user and check if he belongs to the administrators group. We will wrap that in the Is-Administrator function that uses Get-User:

```
PS> function Is-Administrator
{
    $user = Get-User
    $principal = New-Object `
        Security.principal.windowsprincipal($user)
    $principal.IsInRole("Administrators")
}
```

```
PS> Is-Administrator
False
```

Now that we have this tool, we can write our prompt function. Here it comes:

```
PS> function prompt
{
    $path = Get-Location
    $terminator = '$'
    if ((Is-Administrator))
    {
        $terminator = '#'
    }
    "$path $terminator "
}

C:\ $
```

Note how the prompt changes to C:\ \$. It looks like C:\ # in a shell run with administrator credentials.

As bad as side effects that modify the environment can be when caused by a function that is simply supposed to return a value, we can use them in our prompt function to create a richer prompt. The first thing we can do is use color in our prompt. To do that, we have to write the string to the console ourselves and return an empty string, so that the shell does not add anything. Here is a prompt function that prints the prompt in yellow:

```
PS> function prompt
{
    Write-Host ("$(get-location)>") -nonewline `
        -foregroundColor Yellow
    return " "
}
```

We can combine the window manipulation capabilities of the `$host.UI.RawUI` object with prompt evaluation. One of the most interesting ideas floating on the Internet is the way to shorten your prompt string by moving some of the information to the console window title. One of the things that annoy users is seeing long paths that cause the prompt string to get longer than a line and even wrap. We can solve this problem by moving the current path to the window title. This frees up physical space in the prompt string area, and we can list the user name there. Here is our prompt function that does that:

```
function prompt
{
    $user = (Get-User).Name
    $path = Get-Location
    $host.UI.RawUI.WindowTitle = $path
    "$user> "
}
```

Figure 11-4 shows the end result: the current path stays in the window title, and the user name is the only thing in the prompt.



Figure 11-4. A console window showing the current path in the title bar

In conclusion, keep in mind that if your prompt function throws an exception, PowerShell will default to the PS> prompt and will not warn you in any way. To get the real error message, call the prompt function yourself. Here is an example:

```
PS> function prompt
{
    something broken here
}
```

```
PS>prompt
The term 'something' is not recognized as a cmdlet, function, operable
program, or script file. Verify the term and try again.
At line:3 char:14
+     something <<<< broken here
```

Tab Expansion: How Command Completion Works

Many UNIX shells and `cmd.exe` have provided file name completion for quite some time. Again, they did it in a way that reminds me of Henry Ford's saying about Model T car colors—customers can have a car painted any color, so long as it is black. PowerShell goes beyond the uncustomizable completion mechanisms of other shells by using the same approach as with prompt strings—it does its customizations through a global function that can extend tab completion. By default, in addition to file names, hitting the Tab key will complete cmdlet names, cmdlet parameters,

variable names, and object properties. We can extend that by defining a global function and naming it `TabExpansion`. Let's first see how the default function looks:

```
PS> (dir Function:\TabExpansion).Definition
```

```
# This is the default function to use for tab expansi
t handles simple
# member expansion on variables, variable name expans
nd parameter completion
# on commands. It doesn't understand strings so strin
ntaining ; | ( or { may
# cause expansion to fail.

param($line, $lastWord)

& {
    switch -regex ($lastWord)
    {
        # Handle property and method expansion...
    }
}
```

I have omitted a good portion of the function code for the sake of brevity, because we are not really interested in how that code provides completions. As you can probably see from the `param` statement, the function receives two parameters: `$line` holds the entire text line entered so far, and `$lastWord` contains the last word only. The `$lastWord` parameter looks most useful, as it contains the beginning of the word the user is trying to type. The function should return a collection of values with possible completions that the shell will loop over and offer each of them to the user as he presses the Tab key.

Let's try to build something useful. PowerShell does not offer completions on executable names that are in our system PATH. We can fix that by collecting all executable names and writing a better `TabExpansion` function that will complete them. To make matters simple and short, we will get the executables inside the `C:\Windows\System32` folder; all network-related executables that we often use most, like `ping.exe`, are located there. The code gets a bit longer, and we will save it in a separate file, `TabExpand-Utils.ps1`. Here is our first version of the expansion function:

```
#cache the executables to speed up things
$global:systemExes = (dir $env:windir\system32\*.exe) |
    foreach {$_ .Name}

function global:TabExpansion($line, $lastWord)
{
    $result = @()
    #provide expansions for the executables in C:\windows\system32
```

```

foreach ($exe in $systemExes)
{
    if ($exe -match "^$lastWord")
    {
        $result += $exe
    }
}

return $result
}

```

The code needs some explanation. First, the `System32` folder contains *a lot* of files, and getting the list of executables from it is a slow operation. That is why we collect the executable files stored there only once and keep the result in a global variable, `$global:systemExes`. Our `TabExpansion` function is declared as global too—we need that to overwrite the default `TabExpansion` function. The function code loops over all items and checks if they start with the string supplied as the `$lastWord` parameter. We check that by constructing a regular expression on the fly using the `"^$lastWord"` expression to denote that the string starts with the one inside `$lastWord`. If the regular expression matches, that is, if our last word starts with the same character sequence as the current executable name, we add that name to the `$result` collection using the `+=` operator. At the end, we return the possible completions that have been accumulated in `$result`. As a result, words like “pin” get completed to `PING.EXE`:

```

PS> .\TabExpand-Utils.ps1
PS> PING.EXE

```

There is a problem with this approach, though. We lose the default tab expansion by overwriting the default `TabExpansion` function. The easiest way to preserve that is to get the function definition as you saw previously and paste it in our script. This is a bad solution though. What if we need to augment a tab expansion solution that somebody else has implemented? Or what if the next version of PowerShell is incompatible with the default tab expansion code; the next version will contain a new fixed default, but our saved version will break. The solution is to get the default `TabExpansion` function’s script block before overwriting the function and calling the old implementation from our code. Restart your shell session, so that you get back the original `TabExpansion` function, and try the updated version of our code:

```

#save original TabExpansion function call
$global:originalExpansion = (dir Function:\TabExpansion).ScriptBlock
#cache the executables to speed up things
$global:systemExes = (dir $env:windir\system32\*.exe) |
    foreach {$_ .Name}

function global:TabExpansion($line, $lastWord)
{
    $result = @()
    #call the original tab expansion
    $result += &$originalExpansion $line $lastWord
}

```

```

#provide expansions for the executables in C:\windows\system32
foreach ($exe in $systemExes)
{
    if ($exe -match "^\$lastWord")
    {
        $result += $exe
    }
}

return $result
}

```

The difference is in two lines: initializing the `$originalExpansion` global variable to the script block of the original tab expansion function and then calling that script block with the same `$line` and `$lastWord` parameters that we got from PowerShell. This way, we obtain the default completions first before adding our own. As a result, we got our variable name completion back:

```

PS> .\TabExpansion-Utills.ps1
PS> $PROFILE

```

For details on working with script blocks and functions, please refer to Chapters 4 and 5, respectively.

Unfortunately, by defining our own tab expansion mechanism, we seem to lose completions for files in the current folder and completions for cmdlet names. It looks like the shell does complete those names automatically when it detects that the tab expansion has not been overwritten. Fortunately, we can get those completions back by reimplementing them with two loops similar to the one we set up for the system executables. Our code will get somewhat longer, but it will contain three similar loops that hopefully are not too hard to understand. Here is the final version of our function:

```

#save original TabExpansion function call
$global:originalExpansion = (dir Function:\TabExpansion).ScriptBlock
#cache the executables to speed up things
$global:systemExes = (dir $env:windir\system32\*.exe) |
    foreach { $_.Name }

function global:TabExpansion($line, $lastWord)
{
    $result = @()
    #call the original tab expansion
    $result += &$originalExpansion $line $lastWord

    #get items in the current location and provide expansions
    $currentLocationItems = Get-ChildItem * | foreach { $_.Name }
    foreach ($item in $currentLocationItems)
    {

```

```

        if ($item -match "^$lastWord")
        {
            $result += ".$item"
        }
    }

    #get registered cmdlets and aliases and provide expansions
    $commands = (Get-Command -type Alias) + `
        (Get-Command -type Cmdlet) |
        foreach { $_.Name }
    foreach ($command in $commands)
    {
        if ($command -match "^$lastWord")
        {
            $result += $command
        }
    }

    #provide expansions for the executables in C:\windows\system32
    foreach ($exe in $systemExes)
    {
        if ($exe -match "^$lastWord")
        {
            $result += $exe
        }
    }

    return $result
}

```

We have added two loops that go over the child items for the current location and over all commands. We get the command names with the `Get-Command` cmdlet; we call it twice to get the aliases and cmdlets and merge the results into one collection using the `+` operator. The three loops follow the same pattern: test if the current item starts with the characters entered as the `$lastWord` and, if so, add the item to the `$result` collection.

Summary

This chapter introduces the important concepts of shell hosts and custom shells. Providing facilities for easily embedding scripts in various programs is the key feature behind PowerShell that makes it the new system administration tool of choice for many products and solutions. We have gone through the mechanisms that the shell uses to interact with its host and the services the shell host offers to scripts and cmdlets. After that short introduction to the shell environment, we went on to explore how we can configure the most popular of the shell's settings.

The last two sections of the chapter, which deal with prompt strings and tab completions, showed some really advanced customizations. Digging into the harder stuff like tab expansion can be a daunting task at first, but as you saw, it can be simplified to something manageable.

Even if it looks hard, tab expansion customization can tremendously boost your productivity on the command line and is worth looking into. Later, in Chapter 23, we will look into an open source project called PowerTab that customizes tab expansion using the mechanism outlined in this chapter to provide completions for stuff that we had not even thought was possible.



Extending the Type System

In Chapter 1, we barely scratched the surface of the PowerShell type system. We discussed how the built-in object adapters and type extensions work together to decorate existing types coming from all sources. The extended type system is not only an extension to standard .NET, COM, and WMI object types but open for further extension by anyone. We can use it to add properties and members to a live object or modify the type definition so that all objects of that type automatically get the new members. We can add some pretty crazy customizations to the environment by being able to attach custom properties and methods to objects that we use daily. Before learning how to modify objects and types, I used to create a lot of small utility functions that performed frequent operations on objects. I had to use a prefix or a suffix as a part of the function name just to remember what objects it worked on. No more! I have moved all those tiny functions to methods that extend the types in question. I now have shorter names to type and fewer function names to remember.

PowerShell sports a very sophisticated system for turning binary objects into properly formatted text. It allows us to turn a collection of objects into a table, a list, and much more, so that it is easier to read on the screen. The text formatting system has several view types, and it allows us to define custom views that convert an object and its property values into text in the format that we need for a repetitive task. In this chapter, I will go over some of the default view types, and you will learn how to create your own custom views to use and reuse for generating different types of reports.

Modifying Objects and Types

As we already discussed in Chapter 1, a PowerShell object is a complex animal. Whenever you get an object instance, you do not really manipulate the original object no matter where it is coming from: .NET, COM, WMI, or elsewhere. All objects in PowerShell are really instances of the `PSObject` type or its inheritors. The `PSObject` type is the basis—the foundation of the extended type system. It is implemented as an object wrapper, and it holds a reference to the real object. The operations that we perform on the real object all go through the `PSObject` instance, where they can be intercepted and manipulated by the type system. This allows for exposing functionality by adding properties and methods that are not present in the original object. Of course, the opposite is also true: some of the properties and methods on the original object may remain hidden from the script user.

Adding Members to Single Objects

Many script languages keep objects open for modification at all times and allow script authors to easily get an instance of an object and attach additional properties and methods. In some languages, like JavaScript, adding a property is as simple as assigning a property value to an object. If the property is not there, it will be created for us.

PowerShell imposes somewhat stricter rules for adding properties and methods. We can still add them at will to any `PSObject`, but we have to use a special cmdlet for that—`Add-Member`. That cmdlet is the Swiss Army knife for manipulating objects. Ever wanted to have a `FileCount` property on a `DirectoryInfo` object that contains the number of files located inside the folder? Here is how to do it:

```
PS> $currentDir = (Get-Item .)
PS> $currentDir | Add-Member -MemberType ScriptProperty `
-Name FileCount -Value { (dir $this).Count }
PS> $currentDir.FileCount
6
```

The preceding snippet adds a new member to the `$currentDir` object. We call that member a `ScriptProperty`; it appears as a property to callers and is defined by one or two script blocks. One of the blocks is executed when a caller retrieves the property value; the other when he or she sets a new value. Our preceding code snippet defines a read-only property by specifying a getter script block only.

Note The `$this` variable reference is important, because that is how we access the object the property or method was assigned to. In the `DirectoryInfo` example, `$this` refers to our `DirectoryInfo` object. PowerShell always uses that variable to pass the current object instance when attaching methods or properties.

The `Add-Member` cmdlet can modify an object that you pass as the `-InputObject` parameter. Piping objects is usually easier to type, and of course, we can pipe more than one object. Here is how to add the same property to the current folder and the parent one:

```
PS> $dirs = (Get-Item .), (Get-Item ..)
PS> $dirs | Add-Member -MemberType ScriptProperty `
-Name FileCount -Value { (dir $this).Count }
PS> $dirs | select Name,FileCount
```

Name	FileCount
----	-----
12 - Extending the Type System	6
Z:\	28

The `Add-Member` cmdlet can add all sorts of members; the right one depends on the value of the `-MemberType` parameter value. It defines two parameters for passing configuration data: `-Value` and `-SecondValue`. Those two have different uses with respect to the current member type. In this chapter, we will be working with the following member types:

- **AliasProperty:** This property refers to another property on the same object. It is typically used to shorten a property name or to rename a property to make it consistent with other objects. PowerShell uses this type of property quite extensively. For example, the `Process` object has an alias property called `WS` that points to `WorkingSet`, and collections have a `Length` property that returns their count to make them consistent with arrays.
- **NoteProperty:** This property type is similar to public fields in other programming languages. It represents a simple variable storage space that anyone can read from and write to.
- **ScriptProperty:** This property type gets really close to properties as they are implemented in .NET. We have to provide script blocks for the getter and setter functionality of the property. One of them will be called when somebody gets the property value, another, when he or she sets it.
- **ScriptMethod:** This member type defines a method by providing a single script block.

EXTENDING EXTERNAL OBJECTS MAY GET TRICKY

Remember that I said that all type extensions must go through `PSObject`. All object operations go through `PSObject` too, so extending an object must not be a problem, right? Well, it turns out that there are rare occasions where we might have some trouble extending an object. Sometimes, we can get a raw reference to a .NET object, or maybe a WMI one for that matter. We may not immediately detect that we are dealing with a raw object, but we will know that happened if we try to add a member. Raw objects cannot have additional members attached to them.

To illustrate that, let's try to add a `ScriptProperty` member to a string object that will return the first word in the string. Here is our first attempt at doing that:

```
PS> $s = "the quick brown fox"
PS> $s | Add-Member ScriptProperty FirstWord { $this.split()[0] }
PS> $s.FirstWord
PS>
```

Nothing happens. The script code for the property getter looks correct—it splits the string into words and gets the first one. The problem is that our code is not called at all. We need to wrap that string in a `PSObject` before trying to extend it. To do that, it has to be touched by some other PowerShell operation. We have to use the `Add-Member -PassThru` parameter that will make the cmdlet return the modified object and assign the modified value back to our variable. Piping through the cmdlet will transform the string object into a `PSObject` that wraps a string instance. Here is the modified code:

```
PS> $s = "the quick brown fox"
PS> $s = $s | Add-Member ScriptProperty FirstWord {$this.split()[0]} `
-PassThru
PS> $s.FirstWord
the
```

You do not need to worry about converting objects to `PSObject` instances most of the time; any object returned by a cmdlet is already converted for you. Just have the technique in mind, so that you know how to find your way out when you get to that raw object.

Creating New Objects

A little-known fact among PowerShell users is that `PSObject` is a public type and we can create instances of it. What we get back is an empty object with no methods or properties. What good is it, then? Well, combined with the power of `Add-Member`, we can build custom objects. PowerShell does not offer built-in support for defining classes, methods, and properties, but we can simulate that support by creating `PSObject` instances and adding the members we need. This way, we can encapsulate code that works over the same data in a custom object with its own properties and methods. Such objects are especially useful when returning data from a reusable function. This nuisance pops up often. Say we have a function that returns data about different people. How do we return the first and last names of a person? We can use a `Hashtable`, but that would make it inconvenient for the function caller as other cmdlets like `sort` will be hard to use. The best solution is to return a collection of objects that have a `FirstName` and `LastName` property.

To make creating custom objects easier, and adding members quicker, we will build some helper functions. We will introduce the abstraction of a class—a single object that knows how to create instances. Our classes will have a single method, `Create()`, which will create a new instance. First, we need a function that will create our class object. We will call it `Define-Class` and save it to a library script, `Classes-SupportLib.ps1`, which we can reuse from other scripts.

The `Define-Class` function accepts a script block as a parameter—that block will contain our constructor code. It will create a new `PSObject` that will represent our class and will attach a `Note` property containing the constructor. It will then attach the `Create()` method to invoke the constructor. Here is the function code:

```
function Define-Class($constructor)
{
    $class = New-Object PSObject
    $class | Add-Member NoteProperty Constructor $constructor

    $class | Add-Member ScriptMethod Create {
        $instance = New-Object PSObject

        $constructorBlock = $this.Constructor
        $this = $instance
        & $constructorBlock

        return $instance
    }
    return $class
}
```

There is some tricky code in the `Create()` method definition. It first creates a new blank object and then calls the constructor. We need to set the `$instance` value to the `$this` variable, so that the code in the constructor block can refer to the newly created object using the `$this` variable.

Note Keep in mind that modifying the `$this` variable may cause problems, because `$this` already points to the class object inside the `Create()` method. Take extra care when extending this method to save the `$this` variable's original value and restore it after calling the object constructor, if you write code that needs to access the class object.

Now that we can define classes, let's see how we can use them. We need to call `Define-Class`, store the value that it returns as the class object, and use that object to create instances of our class. Let's do that in a demonstration script called `Person-Define.ps1`. Here is its code:

```
. .\Classes-SupportLib.ps1

$Person = Define-Class {
    Write-Host "Creating a Person instance"
}

$p = $Person.Create()
```

Note The `Create()` method looks a lot like static methods or methods defined for all objects of a class. The PowerShell language does not have a statement for creating object instances, and to create objects, we need a way to simulate the `new` keyword that you may have seen in other languages like C#, Visual Basic.NET, or Java. The easiest way to simulate that is to use a static method as our constructor. If you come from a background in the Ruby programming language, you will be familiar with calling the constructor method `New()`; it does the same thing as the Ruby `New()` constructor method.

Note that the script dot-sources the `Classes-SupportLib.ps1` library file. As you can see, we add a simple `Write-Host` statement to our constructor block that will indicate that everything went well. Here is the output from our script:

```
PS> .\Person-Define.ps1
Creating a Person instance
PS>
```

We can now go ahead and extend our script library with functions that can add properties and methods to our class. We want to have functions that we call from within the constructor, so that we encapsulate the entire class definition in a single script block. An additional benefit to that approach is the fact that we do not need to pass the object instance as a parameter to functions that add members; we already have it in the `$this` variable.

We will start with note properties. As we already mentioned, a note property works like a public field, and we can create a new `Add-Field` function. It takes two parameters: `Name` and `Value`. Here is the code:

```
function Add-Field($Name, $Value)
{
    $this | Add-Member NoteProperty $Name $Value
}
```

Let me stress again that the function must be called from within the class constructor, as it relies on the `$this` property being set up for it. We will test our function by calling it from a different script, `Person-NoteProperty.ps1`. Here is the script source:

```
. .\Classes-SupportLib.ps1

$Person = Define-Class {
    Write-Host "Creating a Person instance"

    Add-Field "FirstName" "John"
}

$p = $Person.Create()
$p.FirstName
```

Here is the output we get from our script:

```
PS> .\Person-NoteProperty.ps1
Creating a Person instance
John
```

We can now go on and add some real code to our object: properties and methods. We will create two functions: `Add-Property` and `Add-Method`. The `Add-Method` function takes two parameters: `Name` and `Body`. The `Body` parameter contains a script block with the method body. `Add-Property` is similar with the only difference that it takes two script block parameters, `Getter` and `Setter`, for the two operations that a property may support. Here is the code for those two functions:

```
function Add-Property($Name, $Getter, $Setter)
{
    $this | Add-Member ScriptProperty $Name $Getter $Setter
}

function Add-Method($Name, $Body)
{
    $this | Add-Member ScriptMethod $Name $Body
}
```

We will be implementing a `FullName` property that gets or sets both the `FirstName` and `LastName` properties of our `Person` object. We will also be adding a `Greet()` method that will write a string to the console. We will test our new method and property in a different script file, called `Person-All.ps1`. Here is the code:

```
. .\Classes-SupportLib.ps1

$Person = Define-Class {
    Write-Host "Creating a Person instance"

    Add-Field "FirstName" "John"
    Add-Field "LastName" "Smith"
    Add-Property "FullName" `
    {
        "$($this.FirstName) $($this.LastName)"
    } `
    {
        $value = $args[0]
        $words = $value.Split()
        $this.FirstName = $words[0]
        $this.LastName = $words[1]
    }

    Add-Method "Greet" {
        Write-Host "Hello there. I am $($this.FullName)."
    }
}

$p = $Person.Create()
Write-Host "FullName: $($p.FullName)"
Write-Host "Changing LastName to Caulfield"
$p.LastName = "Caulfield"
Write-Host "FullName: $($p.FullName)"
Write-Host "Changing FullName to Shawn Michaels"
$p.FullName = "Shawn Michaels"
$p.Greet()
```

The most important thing in the preceding code is that we need to use the `$args` special variable to extract parameters for our methods and property setters. Named parameters will not work, and their values will always be `$null`. Here is the output we get when we run our code:

```
PS> .\Person-All.ps1
Creating a Person instance
FullName: John Smith
Changing LastName to Caulfield
FullName: John Caulfield
Changing FullName to Shawn Michaels
Hello there. I am Shawn Michaels.
```

And here is the full code of our `Classes-SupportLib.ps1` script library:

```
function Define-Class($constructor)
{
    $class = New-Object PSObject
    $class | Add-Member NoteProperty Constructor $constructor

    $class | Add-Member ScriptMethod Create {
        $instance = New-Object PSObject

        $constructorBlock = $this.Constructor
        $this = $instance
        & $constructorBlock

        return $instance
    }
    return $class
}

function Add-Field($Name, $Value)
{
    $this | Add-Member NoteProperty $Name $Value
}

function Add-Property($Name, $Getter, $Setter)
{
    $this | Add-Member ScriptProperty $Name $Getter $Setter
}

function Add-Method($Name, $Body)
{
    $this | Add-Member ScriptMethod $Name $Body
}
```

Creating custom objects has become a lot easier with our script library at hand. We can include that library in any other library script and use it to generate collections of our own objects. That should make callers of our scripts much happier.

Adding Members to All Instances of a Class

The `Add-Member` cmdlet is very powerful, but it has one shortcoming: it can only modify a single object. We cannot use it to define a property so that it is available to all objects of the same type. PowerShell has extension methods and properties defined for all objects of a specified type. How does it do that? The answer lies in the `types.ps1xml` file that you can find in PowerShell's installation folder. The folder is typically located at `C:\Windows\System32\WindowsPowerShell\v1.0`. If you open that file and do a quick search for `System.Diagnostics.Process`, you will find the following section:


```

<Type>
  <Name>System.Diagnostics.Process</Name>
  <Members>
    ...
    <AliasProperty>
      <Name>NPM</Name>
      <ReferencedMemberName>NonpagedSystemMemorySize</ReferencedMemberName>
    </AliasProperty>
    <ScriptProperty>
      <Name>Path</Name>
      <GetScriptBlock>$this.Mainmodule.FileName</GetScriptBlock>
    </ScriptProperty>
    ...
  </Members>
</Type>

```

This is how the extended type system is configured. The types.ps1xml file contains a lot of entries similar to the preceding one; many of them are about .NET objects, but there are quite a number of WMI classes too. As you can see from the elements in the XML file, you can define all sorts of members in a similar manner as you would with the `Add-Member` cmdlet. When it loads, PowerShell processes that file and updates its type data. Newly created `PSObject` instances get their member lists from that type data.

What can we do with all that XML and how do we put it to good use? Quite often, I find myself needing security-related information about a file or a number of files: I need the file owner, his or her primary group, and maybe the access rules defined for the file or inherited from folders up the directory tree. To get that information, I always have to get the access control list (ACL) for the file using the `Get-Acl` cmdlet and extract the info from there. We can implement that as three `ScriptProperty` members that we can attach to all `FileInfo` objects.

The easiest way to add our three properties is to just to find the configuration element for the `FileInfo` type and add the new properties: `Owner`, `Group`, and `AccessRules`. That would work, but it is a pretty messy solution. System configuration files are best left intact; imagine what would happen if you had to install a new version of PowerShell a year after you modified your types.ps1xml file? Would you be able to find the exact location of all your changes? Another major problem is portability. To get your type extensions to another computer, you would have to find your original edits in your file and then add them to the new machine's configuration file. Fortunately, PowerShell offers a better way. The cmdlet that can refresh type configuration data, `Update-TypeData`, can include additional XML files. Our solution is to add our type configuration data to a separate file and update the shell type configuration with it. We will call our configuration file `FileSecurity.ps1xml`, because it extends files with security-related information. Here are the contents of the file:

```

<Types>
  <Type>
    <Name>System.IO.FileInfo</Name>
    <Members>
      <PropertySet>
        <Name>FileSecurity</Name>
        <ReferencedProperties>

```

```

        <Name>Name</Name>
        <Name>Owner</Name>
        <Name>Group</Name>
        <Name>AccessRules</Name>
    </ReferencedProperties>
</PropertySet>
<ScriptProperty>
    <Name>Owner</Name>
    <GetScriptBlock>
        (Get-Acl $this).Owner
    </GetScriptBlock>
</ScriptProperty>
<ScriptProperty>
    <Name>Group</Name>
    <GetScriptBlock>
        (Get-Acl $this).Group
    </GetScriptBlock>
</ScriptProperty>
<ScriptProperty>
    <Name>AccessRules</Name>
    <GetScriptBlock>
        (Get-Acl $this).AccessToString
    </GetScriptBlock>
</ScriptProperty>
</Members>
</Type>
</Types>

```

There is nothing too special there, except for the `PropertySet` element. It is not required, but I have added it as a shortcut. I can use its name whenever I need to refer to the properties and want to retrieve all of them.

Note ACL access rules are usually acquired via the `Access` property that returns a collection of `FileSystemAccessRule` or `RegistryAccessRule` objects. They may sometimes get harder to process, so we use the built-in `AccessToString` property that yields a readable string representation.

Now, let's use our new properties. First, we need to update the type configuration by including our file. To do that, we have to pass its path as the `-PrependPath` parameter. Here is how to do that and get one of our properties:

```

PS> Update-TypeData -PrependPath .\FileSecurity.ps1xml
PS> (dir .\FileSecurity.ps1xml).Owner
NULL\Hristo

```

```
PS> (dir .\FileSecurity.ps1xml).Group
NULL\None
PS> (dir .\FileSecurity.ps1xml).AccessRules
BUILTIN\Administrators Allow FullControl
NT AUTHORITY\SYSTEM Allow FullControl
NT AUTHORITY\Authenticated Users Allow Modify, Synchronize
BUILTIN\Users Allow ReadAndExecute, Synchronize
```

Note The type configuration will remain in effect until you restart your shell session. If you want to have those three properties available in all sessions, add the `Update-TypeData` line to your shell profile script.

Now, let's see why we needed that `PropertySet` element in our XML configuration. Property sets are just an easy way to refer to a number of properties. They are a convenience feature that allows us to get all properties at once. Here is how to get a view of our security-related properties by using the `select` command and passing the name of our property set, `FileSecurity`:

```
PS> dir person* | select FileSecurity
```

Name	Owner	Group	AccessRules
----	-----	-----	-----
Person-All.ps1	NULL\Hristo	NULL\None	BUILTIN\Admin...
Person-Define.ps1	NULL\Hristo	NULL\None	BUILTIN\Admin...
Person-NotePro...	NULL\Hristo	NULL\None	BUILTIN\Admin...

Extending Object Formatting

PowerShell commands work with objects and emit objects. At some time though, those objects have to be turned into text, so that they can be displayed to the screen or written to a file. This is what we refer to as object formatting. How is it implemented in PowerShell? The shell designers have come up with the concept of object views. There are different types of views, and an object can have many views associated with it. We can explicitly use a specific view type and display objects as table, list, a wide list, or a custom representation by using one of the formatting cmdlets. The cmdlets that display an object or objects using specific view settings are, respectively, `Format-Table`, `Format-List`, `Format-Wide`, and `Format-Custom`. Each type also has a default view associated with it, and that view is used to display the object when it is not piped through one of the formatting cmdlets. For example, let's display the default view for a list of files:

```
PS> dir *.ps1
```

Directory: Microsoft.PowerShell.Core\FileSystem::Z:\12 - Extending
the Type System

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/10/2007 7:07 PM	693	Classes-SupportLib.ps1
-a---	12/10/2007 7:38 PM	829	Person-All.ps1
-a---	12/10/2007 6:53 PM	127	Person-Define.ps1
-a---	12/10/2007 7:26 PM	181	Person-NoteProperty.ps1

The default view here is the same as the tabular one. Piping the files through `Format-Table` generates the same output:

```
PS> dir *.ps1 | Format-Table
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::Z:\12 - Extending
the Type System
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/10/2007 7:07 PM	693	Classes-SupportLib.ps1
-a---	12/10/2007 7:38 PM	829	Person-All.ps1
-a---	12/10/2007 6:53 PM	127	Person-Define.ps1
-a---	12/10/2007 7:26 PM	181	Person-NoteProperty.ps1

We can format the objects as a list by piping them through `Format-List`:

```
PS> dir *.ps1 | Format-List
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::Z:\12 - Extending
the Type System
```

```
Name           : Classes-SupportLib.ps1
Length          : 693
CreationTime     : 12/10/2007 4:38:10 PM
LastWriteTime    : 12/10/2007 7:07:28 PM
LastAccessTime   : 12/10/2007 4:38:10 PM
VersionInfo      :
...
```

The list output looks more verbose than the table one, but it may be easier to read on some occasions.

In addition to the list view, we can use the wide list one by piping our objects through `Format-Wide`:

```
PS> dir *.ps1 | Format-Wide
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::Z:\12 - Extending
the Type System
```

```
Classes-SupportLib.ps1      Person-All.ps1
Person-Define.ps1          Person-NoteProperty.ps1
```

That view tries to generate output that's as compact as possible, but it may omit some important information.

The `Format-Custom` cmdlet will display, by default, a view that is somewhat similar to a class declaration written in an object-oriented language. It will enumerate all properties and their subproperties and will generate a lot of output. To reduce the output a bit, we will use only one file object:

```
PS> dir .\Classes-SupportLib.ps1 | Format-Custom

class FileInfo
{
    LastWriteTime =
        class DateTime
        {
            DateTime = Monday, December 10, 2007 7:07:28 PM

            ...
            Length = 693
            Name = Classes-SupportLib.ps1
        }
}
```

Creating Our Own Views

The formatting system shares much in common with the type system. It can be extended using XML configuration files. The files have an extension of `.format.ps1xml` and contain view definitions. As with the type system configuration, the stock views that ship with PowerShell are located in the shell installation folder. Here they are:

```
PS> dir $PSHOME\*.format.ps1xml
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\System
32\WindowsPowerShell\v1.0
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	6/30/2007 1:53 PM	22120	Certificate.format.ps1xml
-a---	6/30/2007 1:53 PM	60703	DotNetTypes.format.ps1xml
-a---	6/30/2007 1:53 PM	19730	FileSystem.format.ps1xml
-a---	6/30/2007 1:53 PM	250197	Help.format.ps1xml

```

-a---      6/30/2007  1:53 PM          65283 PowerShellCore.format.ps1xml
-a---      6/30/2007  1:53 PM          13394 PowerShellTrace.format.ps1xm
1
-a---      6/30/2007  1:53 PM          13540 Registry.format.ps1xml

```

Let's try to create a custom tabular view that displays file security information. We already know how to get the owner and the file access rules. We can add the name to that and get a nice-looking table that can help us in our daily security-related tasks. Of course, we would not want to be modifying any of the system format files just to get our view loaded into the shell session. We will use the `Update-FormatData` cmdlet, which works in exactly the same way as `Update-TypeData`.

Creating extra views is a rather under-documented feature of PowerShell. That is why the best way to get started is to take a look at the system views. Since we are now trying to create a tabular view for files, I have just taken the liberty of opening the `FileSystem.format.ps1xml` file that comes from PowerShell and picking a tabular view definition from there. I removed the unneeded parts and added the three security-related properties. Here is the end result, saved in the `FileSecurity.format.ps1xml` file:

```

<Configuration>
  <ViewDefinitions>
    <View>
      <Name>FileSecurity</Name>
      <ViewSelectedBy>
        <SelectionSetName>FileSystemTypes</SelectionSetName>
      </ViewSelectedBy>
      <GroupBy>
        <PropertyName>PSParentPath</PropertyName>
        <CustomControlName>FileSystemTypes-GroupingFormat
        </CustomControlName>
      </GroupBy>
      <TableControl>
        <TableHeaders>
          <TableColumnHeader>
            <Label>Name</Label>
            <Width>20</Width>
            <Alignment>left</Alignment>
          </TableColumnHeader>
          <TableColumnHeader>
            <Label>Owner</Label>
            <Width>12</Width>
            <Alignment>left</Alignment>
          </TableColumnHeader>
          <TableColumnHeader>
            <Label>AccessRules</Label>
            <Alignment>left</Alignment>
          </TableColumnHeader>
        </TableHeaders>

```

```

<TableRowEntries>
  <TableRowEntry>
    <Wrap/>
    <TableColumnItems>
      <TableColumnItem>
        <PropertyName>Name</PropertyName>
      </TableColumnItem>
      <TableColumnItem>
        <ScriptBlock>
          (Get-Acl $_).Owner
        </ScriptBlock>
      </TableColumnItem>
      <TableColumnItem>
        <ScriptBlock>
          (Get-Acl $_).AccessToString
        </ScriptBlock>
      </TableColumnItem>
    </TableColumnItems>
  </TableRowEntry>
</TableRowEntries>
</TableControl>
</View>
</ViewDefinitions>
</Configuration>

```

A table view definition has three important parts:

- *View name:* We will use the view name when we instruct Format-Table to format files using our view.
- *ViewSelectedBy:* This element defines the types this view applies to. The `FileSystemTypes` setting you see there is defined in the `FileSystem.format.ps1xml` file, and we just reuse the value here.
- *GroupBy:* This property will be used to group objects together. When it comes to files and folders, the `PSParentPath` property is the best option, as it groups together files and folders that belong to the same folder.
- *TableControl:* This is the actual table definition. It contains settings for the table headers and for the table rows.

Our table has three columns: Name, Owner, and AccessRules. We have used the same script code to retrieve the column values as we did in our type extension example. Let's now update the shell format data with our view definition and try out the view. To use our view, we have to send some files to the Format-Table cmdlet and specify the view parameter with a value of `FileSecurity`:

```
PS> Update-FormatData -PrependPath .\FileSecurity.format.ps1xml
PS> dir person* | Format-Table -view FileSecurity
```

Directory: Microsoft.PowerShell.Core\FileSystem::Z:\12 - Extending the Type System

Name	Owner	AccessRules
----	-----	-----
Person-All.ps1	NULL\Hristo	BUILTIN\Administrators Allow FullControl NT AUTHORITY\SYSTEM Allow FullControl NT AUTHORITY\Authenticated Users Allow Modify, Synchronize BUILTIN\Users Allow ReadAndExecute, Synchronize
Person-Define.ps1	NULL\Hristo	BUILTIN\Administrators Allow FullControl NT AUTHORITY\SYSTEM Allow FullControl NT AUTHORITY\Authenticated Users Allow Modify, Synchronize BUILTIN\Users Allow ReadAndExecute, Synchronize
...		

Summary

In this chapter, you have learned how to extend both the type system and the object formatting system. While extending those two systems may seem like the blackest of PowerShell magic, doing so can certainly offer significant benefits. Using type extensions, we can create very readable scripts that are a pleasure to extend and maintain. Formatting extensions can help you create the right view that will generate that report you need to produce every day. The possibilities are limitless.

One last word of warning—PowerShell gives us some really powerful tools that can cause chaos if used all the time without proper judgment. It is easy to turn all your utility scripts into extension methods, use them all the time, and forget that they are extension methods. You will learn to depend on those extensions, and that dependence will break your scripts when they run on other machines that may not have your helper extensions. Be careful in the ways you extend the system.



Getting Help

Let's face it—reading a book will not automatically make you an expert on a subject no matter how good the book is. Going through a good book is like getting a good map that will guide you through your journey on the road to mastery. The map can help you overcome obstacles and move faster, but it will not walk the path for you. You will not attain thorough knowledge and skill until you start doing what you just read about. Of course, there will be hard moments when you hit those undocumented obstacles and get lost on your journey. This is where your most important lesson has to kick in—you must know where to look for help and how to explore the environment, so that you can learn things on your own.

One of PowerShell's greatest strengths is that it is a fully interactive environment. It is so consistent in its object orientation that everything is an object, and you can literally grab a reference to anything—without any regard for whether it is an intrinsic part of the shell, a .NET object, a COM object or something else—and look at it, its properties, and its methods. Sometimes, the exploration process feels like a role-playing game where you get to stuff maps and all sorts of other things in your inventory and try to use them everywhere until something clicks and you make some progress in your quest. Learning a new command or getting help on using that obscure parameter that you see for the first time is just a matter of poking around with the help system until you get things working. Our goal for this chapter is to go through the essential learning tools that can get us out when we are stuck. We will learn how to use the built-in help, examine objects to get more information about them, explore new commands, and even create our own helpers that can search the web for additional information.

The Help System

The PowerShell designers seem to have known from the very beginning that if the product was to be adopted and really used by a large number of people, it had to be easy to learn. Until now, we have concentrated on the conventions and the descriptive names for all built-in commands that read like English. Of course, even those can sometimes be difficult and that is why every command is fully documented. You can get help by calling the `Get-Help` cmdlet. `Get-Help`, being a good cmdlet citizen, returns objects, which may not be the best thing to spew at a novice user. To make the help text more accessible, PowerShell also provides a `help` command, which is a function that just calls `Get-Help` and pipes its output through the `more.com` utility so that users get paging when they read a larger body of text.

The Basics

The simplest form of calling for help is typing **help** at the console prompt. It will list the available help topics:

```
PS> help
```

Name	Category	Synopsis
----	-----	-----
ac	Alias	Add-Content
asnp	Alias	Add-PSSnapin
clc	Alias	Clear-Content
cli	Alias	Clear-Item
clp	Alias	Clear-ItemProperty
clv	Alias	Clear-Variable
cpi	Alias	Copy-Item
...		

You can provide a wildcard that will filter the topics. For example here is how to display the language-specific help topics by relying on the fact that their names all start with the “about_” string:

```
PS> help about_*
```

Name	Category	Synopsis
----	-----	-----
about_Alias	HelpFile	Using alternate nam...
about_Arithmetic_Ope...	HelpFile	Operators that can ...
about_Array	HelpFile	A compact data stru...

You can call help and pass the name of the topic you want displayed:

```
PS> help ac
```

NAME

Add-Content

SYNOPSIS

Adds content to the specified item(s).

...

The preceding example uses the ac command, which is an alias for the Add-Content cmdlet. That means that getting help ac is the same as getting help for Add-Content:

```
PS> help Add-Content
```

NAME

Add-Content

SYNOPSIS

Adds content to the specified item(s).

...

PowerShell automatically adds help topics for all aliases that have been declared. This both saves typing and provides help to users who do not need to know the cmdlet that an alias points to.

As you might have seen from the previous help topic list, every topic has an associated category. To get all category values for the built-in topic, we can use the `Get-Help` cmdlet. Unlike the `help` command, it will return the actual help topic objects. We will pipe the collection through `select` and get the unique category values:

```
PS> Get-Help * | select Category -Unique
```

```
Category
```

```
-----
```

```
Alias
```

```
Cmdlet
```

```
Provider
```

```
HelpFile
```

There we go—the built-in topics have the following category values:

- **Alias:** As you have already seen, this is a topic automatically created for all aliases.
- **Cmdlet:** This is a topic documenting how a cmdlet, either built-in or coming from a third-party snap-in, works.
- **Provider:** This topic documents an installed provider. Out of the box, PowerShell ships with documentation for its built-in providers: `Alias`, `Environment`, `FileSystem`, `Function`, `Registry`, `Variable`, and `Certificate`, and a well-behaved custom provider should add topics to that category.
- **HelpFile:** This is a concept topic. Those are the `about_*` topics that discuss specific language features like branching, looping, variables, and so on.
- **Get-Help:** This topic can retrieve topics based on the category name. Here is how we can get the language-specific topics or the ones of the `HelpFile` category:

```
PS> Get-Help -Category HelpFile
```

Name	Category	Synopsis
----	-----	-----
<code>about_Alias</code>	<code>HelpFile</code>	Using alternate nam...
<code>about_Arithmetic_Ope...</code>	<code>HelpFile</code>	Operators that can ...
<code>about_Array</code>	<code>HelpFile</code>	A compact data stru...
...		

`Get-Help` and `help` support another important parameter that controls the amount of text that gets spewed back to you. You can control that using the `-detailed`, `-full`, and `-examples` switches. Here are the possible configurations:

- *No switches (the default)*: You will get a short, about one page long, explanation of what the command does. It will include a list of the supported parameters, a brief description, and one or two examples.
- *-detailed*: You will get a longer explanation that will contain a list of all parameters with brief descriptions about what every one of them does. You will also get a list of examples of the command in action.
- *-full*: This gets you absolutely everything. You will get full information about all the parameters: if they are required, their possible values, and so on. The list of examples will contain additional information about each example, including details on what the code does and how it accomplishes the task at hand.
- *-examples*: Passing this switch indicates that you need the examples only. You will get the full text for all examples, including the titles, descriptions, and the sample code. No other information about the command will be returned.

Parameter Details

Very often, just one of the parameters of a command really interests you. You should not have to wade through pages and pages of text just to get to the section that interests you. Get-Help supports the `-parameter` parameter. You can use it to get help about the specific parameter that you are trying to use. Here is how to get information about the `-filter` parameter of the `Get-ChildItem` cmdlet:

```
PS> Get-Help Get-ChildItem -parameter filter
```

```
-filter <string>
```

Specifies a filter in the provider's format or language. The value of this parameter qualifies the `Path` parameter. The syntax of the filter, including the use of wildcards, depends on the provider. Filters are more efficient than other parameters, because the provider applies them when retrieving the objects, rather than having Windows PowerShell filter the objects after they are retrieved.

Required?	false
Position?	2
Default value	
Accept pipeline input?	false
Accept wildcard characters?	True

You get a short description of what the parameter does and a formatted list with the parameter specification: if it is mandatory (required), its position (if it is a positional parameter), the default value, if the pipeline input is interpreted as the parameter value and if the parameter supports wildcards. Let's try something wild: getting help about the `Get-Help` cmdlet `Parameter` parameter:

```
PS> Get-Help Get-Help -parameter Parameter
```

```
-parameter <string>
```

Displays a detailed description of the specified parameter. These descriptions are included in the Full view of help. Wildcards are permitted.

```
Required?                false
Position?                named
Default value
Accept pipeline input?   false
Accept wildcard characters? True
```

I sometimes like to think about PowerShell as self-documenting. As you saw from the preceding example, we can use wildcards to get information about several parameters. We can use that to get to all the parameters by providing `*` as the parameter name:

```
PS> Get-Help Get-ChildItem -parameter *
```

```
-path <string[]>
```

Specifies a path to one or more locations. Wildcards are permitted. The default location is the current directory (`.`).

```
Required?                false
Position?                1
Default value            <NOTE: if not specified uses the Current location>
Accept pipeline input?   true (ByValue, ByPropertyName)
Accept wildcard characters? True
```

```
...
```

Using wildcards in parameter names comes in very handy when we are not sure about the parameter name, or we are just feeling lazy and do not want to type the entire word:

```
PS> Get-Help Get-ChildItem -parameter fo*
```

```
-force <SwitchParameter>
```

Overrides restrictions that prevent the command from succeeding, just so the changes do not compromise security. For example, Force will override the read-only attribute or create directories to complete a file path, but it will not attempt to change file permissions.

```
Required?                false
Position?                named
Default value            False
Accept pipeline input?   false
Accept wildcard characters? False
```

Advanced Techniques

If you are like me, you love digging deep and finding out how things really work. The PowerShell help system is far more advanced than just a bunch of text files thrown together that you are allowed to read. The advanced queries and the wealth of examples and detail you can get on parameters and other command attributes all speak of an unusual help system. The compiled HTML help files or *.chm files as most users refer to them, are not up to the challenge of providing such experience to users. The PowerShell help system relies on another technology that is a part of the Windows Vista assistive technology. The help contents is stored in an XML-based file format that was originally called MAML, short for Microsoft Assistive Markup Language, and was later renamed to just Assistive Markup Language (AML). I will not go into AML details here but will just show how a typical help file looks.

By convention, PowerShell requires that the help file for a snap-in DLL be named similarly to the DLL name. This convention should be followed by cmdlet authors, and of course, the built-in cmdlet DLL's already follow it. Let's now look at the help file for the Microsoft.PowerShell.Management snap-in that contains the cmdlets dealing with files and items. Let's get the snap-in first:

```
PS> Get-PSSnapin Microsoft.PowerShell.Management
```

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

Where is the .NET assembly, the DLL file, that hosts the cmdlet classes? Let's get it from the ModuleName property:

```
PS> (Get-PSSnapin Microsoft.PowerShell.Management).ModuleName
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell.Commands.Management.dll
```

By convention, the name of the help file is Microsoft.PowerShell.Commands.Management.dll-Help.xml. It is localized and can be found either next to the DLL file or in a subfolder having the culture name. That setup allows for providing different versions for the same help file, localized for different cultures. The default en-US version of the help file is inside C:\Windows\system32\windowspowershell\v1.0\en-US\ on 32-bit Windows Vista machines.

So, what does a typical help file contain? Let's open the preceding file and take a look at the documentation for the Add-Content cmdlet. The content is wrapped in a <command:command> XML tag and looks like this:

```
<command:command
xmlns:maml="http://schemas.microsoft.com/maml/2004/10"
xmlns:command="http://schemas.microsoft.com/maml/dev/command/2004/10"
xmlns:dev="http://schemas.microsoft.com/maml/dev/2004/10">
  <command:details>
    <command:name>
      Add-Content
    </command:name>
```

```

    <maml:description>
      <maml:para>
        Adds content to the specified item(s).
      </maml:para>
    </maml:description>
    <maml:copyright>
      <maml:para></maml:para>
    </maml:copyright>
    <command:verb>add</command:verb>
    <command:noun>content</command:noun>
    <dev:version></dev:version>
  </command:details>
  <maml:description>
    <maml:para>The Add-Content cmdlet appends content to a
    specified item or file. You can specify the content by typing
    the content in the command or by specifying an object that
    contains the content.</maml:para>
  </maml:description>
  ...
</command:command>

```

Why do we need to look at the XML file? PowerShell works with objects, and Get-Help returns objects that we can use. Guess what? The objects we get back are constructed from that AML document, and they are adapted by the XML type adapter. That means we can access any property we wish, get detailed information, and build queries not directly supported by the Get-Help cmdlet. Here is how we get the help topic object using Get-Help:

```

PS> $h = Get-Help Add-Content
PS> $h

```

```

NAME
    Add-Content

```

```

SYNOPSIS
    Adds content to the specified item(s).
    ...

```

Get-Help really does not format any text. It returns `MamlCommandHelpInfo` objects and the object formatting infrastructure takes care of the rest. The `MamlCommandHelpInfo` has an associated custom formatting view that is defined in the same way we defined our custom object view in Chapter 12. If you are interested in details about the view definition, review the `Help.format.ps1xml` formatting file in the PowerShell installation folder.

Back to AML and help topic properties—let's get the details and the description for our `Add-Content` help topic:

```
PS> $h.details
```

```
NAME
```

```
Add-Content
```

```
SYNOPSIS
```

```
Adds content to the specified item(s).
```

```
PS> $h.description
```

The Add-Content cmdlet appends content to a specified item or file. You can specify the content by typing the content in the command or by specifying an object that contains the content.

Need the parameters? No problem:

```
PS> $h.parameters
```

```
-path <string[]>
```

Specifies the path to the items that receive the additional content. Wildcards are permitted. If you specify multiple paths, use commas to separate the paths.

Required?	true
Position?	1
Default value	N/A - The path must be specified
Accept pipeline input?	true (ByPropertyName)
Accept wildcard characters?	true

You can get statistics about the parameters or access them by index:

```
PS> $h.parameters.parameter.length
```

```
11
```

```
PS> $h.parameters.parameter[4]
```

```
-value <Object[]>
```

Specifies the content to be added. Type a quoted string, such as "This data is for internal use only" or specify an object that contains content, such as the DateTime object that Get-Date generates.

You cannot specify the contents of a file by typing its path, because the path is just a string, but you can use a Get-Content command to get the content and pass it to the Value parameter.


```

Required?                true
Position?                2
Default value
Accept pipeline input?    true (ByValue, ByPropertyName)
Accept wildcard characters? false

```

Why the awkward notation of `$h.parameters.parameter`? The AML file explains it all. Here is the parameters declaration:

```

<command:parameters>
  <command:parameter required="true" variableLength="false"
    globbing="true" pipelineInput="true (ByPropertyName)"
    position="1">
    <maml:name>path</maml:name>
    <maml:description>
      <maml:para>Specifies the path to the items that receive
        the additional content. Wildcards are permitted. If you
        specify multiple paths, use commas to separate the
        paths.</maml:para>

      </maml:description>
      <command:parameterValue required="true"
        variableLength="false">
        string[]
      </command:parameterValue>
      <dev:type>
        <maml:name>string[]</maml:name>
        <maml:uri/>
      </dev:type>
      <dev:defaultValue>
        N/A - The path must be specified
      </dev:defaultValue>
    </command:parameter>
    ...
  </command:parameters>

```

The XML object adapter turns the `<command:parameters>` tag into a `parameters` property, and all the `<command:parameter>` tags get converted into a collection.

Let's now look at how examples are stored in an AML file. Here is the first Add-Content example:

```

<command:examples>
  <command:example>
    <maml:title>
      ----- EXAMPLE 1 -----
      -----
    </maml:title>

```

```

    <maml:introduction>
        <maml:para>C:\PS></maml:para>
    </maml:introduction>
    <dev:code>add-content -path *.txt -exclude help* -value &quot;
        END&quot;</dev:code>
    <dev:remarks>
        <maml:para>This command adds &quot;END&quot; to all text
            files in the current directory, except for those with file
            names that begin with &quot;help&quot;.</maml:para>
    </dev:remarks>
</command:example>
...
</command:examples>

```

This means we can access the examples in the same manner that we used for parameters:

```

PS> $h.examples.example.length
3
PS> $h.examples.example[0]

```

----- EXAMPLE 1 -----

```

C:\PS>add-content -path *.txt -exclude help* -value "END"

```

This command adds "END" to all text files in the current directory, except for those with file names that begin with "help".

Being able to access the different help topic subobjects is a very powerful technique. We can use it to create a script that will search all cmdlet help examples for a specific string or to get all sorts of examples: Get-/Set-Content examples that work with text files, examples that work on a range of files using foreach, and so on. Here is one possible implementation of a sample code search utility:

```

param ($Name = "*", $Query = { $true })

$topics = Get-Help $Name

foreach ($topic in $topics)
{
    $matchingExamples = $topic.examples.example | where $Query
    if ($matchingExamples)
    {
        #output the found examples
        $matchingExamples
    }
}

```

You can find the code in the `Search-Examples.ps1` script. We declare two parameters: `$Name`, which is the cmdlet name filter, and `$Query`, a script block that will be evaluated for every example and will serve as our criteria for finding examples. Our code gets all help topics that `Get-Help` returns given our name filter. It then applies the query script block to all examples and returns only those that have evaluated to `$true`. You can read more about script blocks in Chapter 4 and about the script-block-as-query technique in Chapter 5. Here is how we use our script to get examples for the `Get-Content` and `Set-Content` cmdlets (using the `*content*` filter) that works over a collection of files using `foreach`:

```
PS> .\Search-Examples.ps1 -Name *content* -Query `
    { $_.code -like "*foreach*" }
```

----- EXAMPLE 3 -----

```
C:\PS>(get-content Notice.txt) | foreach-object {$_ -replace "Warning"
, "Caution"} | set-content Notice.txt
```

This command replaces all instances of "Warning" with "Caution" in the `Notice.txt` file.

It uses the `Get-Content` cmdlet to get the content of `Notice.txt`. The pipeline operator sends the results to the `ForEach-Object` cmdlet, which applies the expression to each line of content in `Get-Content`. The expression uses the `"$_"` symbol to refer to the current item and the `Replace` parameter to specify the text to be replaced.

...

Looking at the output returned by the script, you can see that we got back the third example in the `Get-Content` cmdlet help topic. This is the only one that uses `foreach` in its code.

Passing a script block as the query allows for greater flexibility in our search criteria. What if you forget the name of an example, but you still remember it explained something about a "notice.txt" file in its remarks? Here's how you can get that example:

```
PS> .\Search-Examples.ps1 -Name *content* -Query `
    { $_.remarks -like "*notice.txt*" }
```

----- EXAMPLE 3 -----

```
C:\PS>(get-content Notice.txt) | foreach-object {$_ -replace "Warning"
, "Caution"} | set-content Notice.txt
```

This command replaces all instances of "Warning" with "Caution" in the Notice.txt file.

It uses the Get-Content cmdlet to get the content of Notice.txt. The pipeline operator sends the results to the ForEach-Object cmdlet, which applies the expression to each line of content in Get-Content. The expression uses the "\$_" symbol to refer to the current item and the Replace parameter to specify the text to be replaced.

...

Getting Command Information

PowerShell can invoke all types of commands: cmdlets, functions, aliases, script files, external applications, and even external files that have a registered default action with the windows shell. The most important tool for working with commands is the Get-Command cmdlet. It provides information about registered cmdlets, accessible external applications, and files. You can use it to group related commands and discover new commands that you do not know about. This is where the PowerShell verb-noun convention shines: if you know a cmdlet that works with a specific type of object, you can get all commands that have the same noun in their name, which makes it quite easy to cover all operations about an object. Say, for example, that you know and use the Rename-Item and Move-Item cmdlets to rename and move files respectively, and you want to expand your knowledge to cover other operations. Let's get all cmdlets that work with items:

```
PS> Get-Command -noun *item*
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Clear-Item	Clear-Item [-Path] <Str...
Cmdlet	Clear-ItemProperty	Clear-ItemProperty [-Pa...
Cmdlet	Copy-Item	Copy-Item [-Path] <Stri...
Cmdlet	Copy-ItemProperty	Copy-ItemProperty [-Pat...
Cmdlet	Get-ChildItem	Get-ChildItem [[-Path] ...
Cmdlet	Get-Item	Get-Item [-Path] <Strin...
Cmdlet	Get-ItemProperty	Get-ItemProperty [-Path...
Cmdlet	Invoke-Item	Invoke-Item [-Path] <St...
Cmdlet	Move-Item	Move-Item [-Path] <Stri...
Cmdlet	Move-ItemProperty	Move-ItemProperty [-Pat...
Cmdlet	New-Item	New-Item [-Path] <Strin...
Cmdlet	New-ItemProperty	New-ItemProperty [-Path...
Cmdlet	Remove-Item	Remove-Item [-Path] <St...
Cmdlet	Remove-ItemProperty	Remove-ItemProperty [-P...
Cmdlet	Rename-Item	Rename-Item [-Path] <St...
Cmdlet	Rename-ItemProperty	Rename-ItemProperty [-P...
Cmdlet	Set-Item	Set-Item [-Path] <Strin...
Cmdlet	Set-ItemProperty	Set-ItemProperty [-Path...

Now, we have all cmdlets that create, delete, modify, and otherwise amend both items and subitems. The next logical step is to review the help topics for each of the cmdlets returned.

We can do the same for the cmdlet verb. Let's get all cmdlets that stop something:

```
PS> Get-Command -verb *stop*
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Stop-Process	Stop-Process [-Id] <Int...
Cmdlet	Stop-Service	Stop-Service [-Name] <S...
Cmdlet	Stop-Transcript	Stop-Transcript [-Verbo...

Now, we know where to look if we want to stop a process or a service. The transcript is a way to log your console session, so you know how to stop it now, and chances are you already guess how to start it too—using the Start-Transcript cmdlet.

We can search for cmdlets by name, and that includes both the noun and verb parts. The Name parameter is the default one, and we do not need to specify it. Let's now get all commands dealing with certificates:

```
PS> Get-Command *certificate*
```

CommandType	Name	Definition
-----	----	-----
Application	Certificate.format.ps1xml	C:\Windows\System32\Win...
Cmdlet	Get-PfxCertificate	Get-PfxCertificate [-Fi...

We got the Get-PfxCertificate cmdlet and an external application, the Certificate.format.ps1.xml file. You could argue that the Application command type is not suitable for an XML file; it is not a runnable file anyway. PowerShell uses the Application command type to denote any file that can be opened by the operating system. That includes executables (real applications) and files that have a registered Open action with Windows Explorer. Typing the Certificate.format.ps1xml name at the console will execute its default Windows shell action, which is opening the file in notepad.exe.

We can use Get-Command to learn more about what happens when we type a specific command at the console prompt. Here is how to use it to learn about several commands:

```
PS> Get-Command help
```

CommandType	Name	Definition
-----	----	-----
Function	help	param([string]\$Name,[st...
Application	help.exe	C:\Windows\system32\hel...

```
PS> Get-Command c:
```

CommandType	Name	Definition
-----	----	-----
Function	C:	Set-Location C:

PS> Get-Command ps

CommandType	Name	Definition
-----	----	-----
Alias	ps	Get-Process

As you can see, the help command is a function, the C: command that you use to go to the C: drive is a function too—it calls the Set-Location cmdlet to go to the C: drive. The ps command is an alias for Get-Process, and there is an external application help.exe in C:\Windows\System32 that we can run if needed.

The Definition property is really useful for external commands. I do not remember how many times I have wondered where the executable file for a given command resides. No more of that! Here is how to get the path to several well known Windows utilities:

```
PS> (Get-Command ping).Definition
C:\Windows\system32\PING.EXE
PS> (Get-Command tracert).Definition
C:\Windows\system32\TRACERT.EXE
PS> (Get-Command calc).Definition
C:\Windows\system32\calc.exe
PS> (Get-Command winword).Definition
```

Getting Information About Objects

Quite often, we get an unidentified object from a cmdlet or another object. We were in that situation when we were working with AML objects previously. What can we do to get information about such objects? The answer is the Get-Member cmdlet. Get-Member lists an object’s members: its properties, methods, fields, and so on. The easiest way to use it is to just pipe your objects through the cmdlet. Here is how we get a file item’s properties:

PS> Get-Item test.txt | Get-Member

TypeName: System.IO.FileInfo

Name	MemberType	Definition
----	-----	-----
AppendText	Method	System.IO.StreamWriter App...
CopyTo	Method	System.IO.FileInfo CopyTo(...
Create	Method	System.IO.FileStream Create()
CreateObjRef	Method	System.Runtime.Remoting.Obj...

```

CreateText      Method      System.IO.StreamWriter Cre...
Decrypt         Method      System.Void Decrypt()
Delete          Method      System.Void Delete()
Encrypt         Method      System.Void Encrypt()
...

```

The output conveniently starts with the object type name and goes on to list the various members. We can restrict it to showing only properties; these are what we need most often:

```
PS> Get-Item test.txt | Get-Member -MemberType Property
```

```
TypeName: System.IO.FileInfo
```

Name	MemberType	Definition
Attributes	Property	System.IO.FileAttributes Attributes {g...
CreationTime	Property	System.DateTime CreationTime {get;set;}
CreationTimeUtc	Property	System.DateTime CreationTimeUtc {get;s...
Directory	Property	System.IO.DirectoryInfo Directory {get;}
DirectoryName	Property	System.String DirectoryName {get;}
Exists	Property	System.Boolean Exists {get;}
...		

We can get static members too by using the `-static` switch parameter. Here is how to get the static properties of the `DateTime` type:

```
PS> [datetime] | Get-Member -static -MemberType Property
```

```
TypeName: System.DateTime
```

Name	MemberType	Definition
MaxValue	Property	static System.DateTime MaxValue {get;set;}
MinValue	Property	static System.DateTime MinValue {get;set;}
Now	Property	System.DateTime Now {get;}
Today	Property	System.DateTime Today {get;}
UtcNow	Property	System.DateTime UtcNow {get;}

We can pass both a type and an object instance when we need static members; `Get-Member` is smart enough to get the type itself:

```
PS> [datetime]::Now | Get-Member -static -MemberType Property
```

```
TypeName: System.DateTime
```

Name	MemberType	Definition
MaxValue	Property	static System.DateTime MaxValue {get;set;}
MinValue	Property	static System.DateTime MinValue {get;set;}
Now	Property	System.DateTime Now {get;}
Today	Property	System.DateTime Today {get;}
UtcNow	Property	System.DateTime UtcNow {get;}

We can use `Get-Member` as a low-cost help browser and get the signature for an object method. For example, this is how we can verify that the `[datetime]::Parse` static method accepts a string parameter and returns a `DateTime` object:

```
PS> [datetime] | Get-Member -name Parse -static
```

```
TypeName: System.DateTime
```

Name	MemberType	Definition
Parse	Method	static System.DateTime Parse(String s), static Sys...

```
PS> [datetime]::Parse("1/1/2010")
```

```
Friday, January 01, 2010 12:00:00 AM
```

Of course, that is not a substitute for reading the documentation for the .NET object in MSDN. The next section will show us how to easily get to that.

Using the Internet to Get Help

Quite often, the integrated help is not enough for our needs, particularly if we interact with an external system that is documented elsewhere. .NET objects are not documented in PowerShell; their reference material is available on the Microsoft Developer Network (MSDN) web site for example. Fortunately, PowerShell makes it quite easy to create several global functions that we can use to open a web browser window and get help about an object.

Note The Microsoft Developer Network, or MSDN for short, is the Microsoft division responsible for managing the company's relationships with software developers. It offers services for distributing software tools, provides a home for many blogs and several online communities, and hosts huge documentation resources related to developer tools and products. Many objects we are manipulating with PowerShell are really .NET objects, and those are documented on the MSDN site. The site start page can be found at <http://msdn.microsoft.com>.

There are many ways to open a web site from within PowerShell. The most common one is to launch a browser instance and pass the URL as a command-line parameter. Another

approach is to instantiate the Internet Explorer COM object and use its methods to navigate. The easiest and probably the best way for me is to use the Windows Script Host Shell.Application COM object and open a URL with its `Open()` method. This will use the default web browser on the machine to open the web address. We could have just passed the URL as a command-line parameter to `ieexplore.exe`, the Internet Explorer executable. We use the Shell COM object as a courtesy to those who use other browsers, such as Mozilla Firefox. I will cover working with COM objects in Chapter 20; for now, just know that the function below opens a web browser window using a COM object that comes installed with Windows.

```
function Open-Url($url)
{
    $shell = New-Object -ComObject Shell.Application
    if ($url -notlike "http://*")
    {
        $url = "http://" + $url
    }
    $shell.Open($url)
}
```

For convenience, the function adds an `http://` prefix to the URL if it has not been provided by the user. Here is how we can open the MSDN web site:

```
PS> Open-Url "msdn.microsoft.com"
PS>
```

Figure 13-1 shows the result using Mozilla Firefox as the default web browser.



Figure 13-1. *The MSDN site opened from within PowerShell*

The next step is to create a `Search-Msdn` function that takes a single parameter and opens it as a MSDN search URL. Here is the code:

```
function Search-Msdn($query)
{
    $encodedQuery = Encode-Url $query
    Open-Url ("http://search.msdn.microsoft.com/search/" + `
        "Default.aspx?query=$encodedQuery")
}
```

Passing all sorts of strings in URLs requires that we encode them using a standard URL-encode algorithm. We do that using the `System.Web.HttpUtility` .NET class. Here is how we can wrap it in a function that loads the `System.Web` assembly and calls the class:

```
function Encode-Url($inputString)
{
    $null = [Reflection.Assembly]::LoadWithPartialName("System.Web")
    return [Web.HttpUtility]::UrlEncode($inputString)
}
```

Let's use the new function to get some help on the `String.Split` method:

```
PS> Search-Msdn "String.Split"
PS>
```

Figure 13-2 shows the result displayed in the browser.

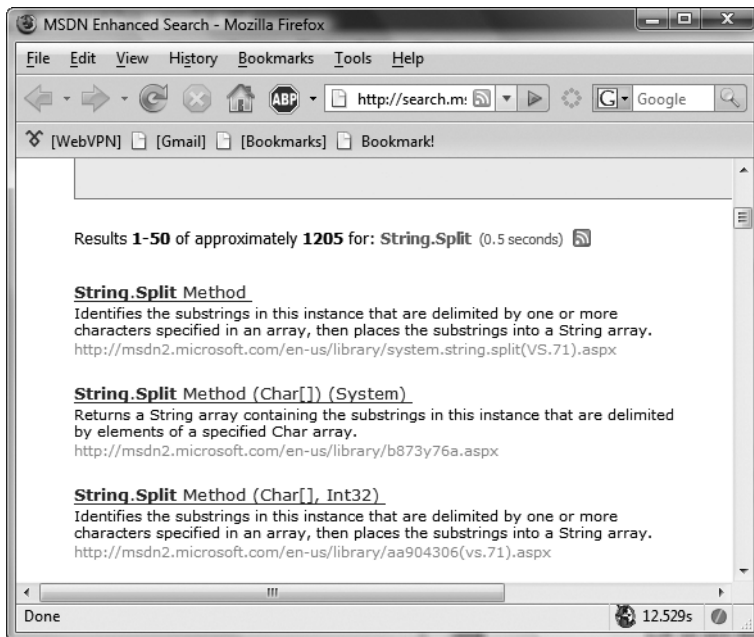


Figure 13-2. *The MSDN search results view*

Since we are working with objects, we often get their type and then search for that. Life would be a lot easier if we had a function that gets an object and opens the MSDN help page for its type. We can do that by getting the object type and using it to craft a URL that MSDN understands. Here is our `Show-MsdnHelp` function:

```
function Show-MsdnHelp($object)
{
    $typeName = $object.GetType().FullName
    Open-Url "http://msdn2.microsoft.com/en-us/library/$typeName.aspx"
}
```

Let's use that function to get information about the objects returned by the `Get-Process` cmdlet:

```
PS> Show-MsdnHelp (Get-Process)[0]
PS>
```

Figure 13-3 shows the browser window displaying the `Process` class MSDN help page.

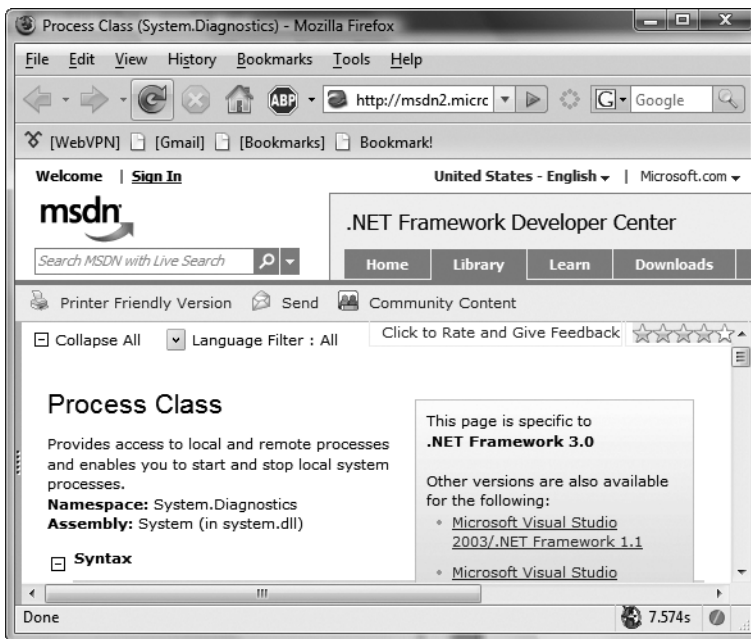


Figure 13-3. *The MSDN help page for the `System.Diagnostics.Process` class*

Sometimes using MSDN is not enough, and we need a broader search. Since we are in the shell window, typing a search term in there to open a browser window to search using the Google or Live.com search engines would be the quickest solution. Here is how to write a `Search-Google` function:

```
function Search-Google($query)
{
    $encodedQuery = Encode-Url $query
    Open-Url "http://www.google.com/search?q=$encodedQuery"
}
```

Let's use it to search for a book on Windows PowerShell:

```
PS> Search-Google '"Pro Windows PowerShell"'
PS>
```

Figure 13-4 shows the browser window containing the search results.

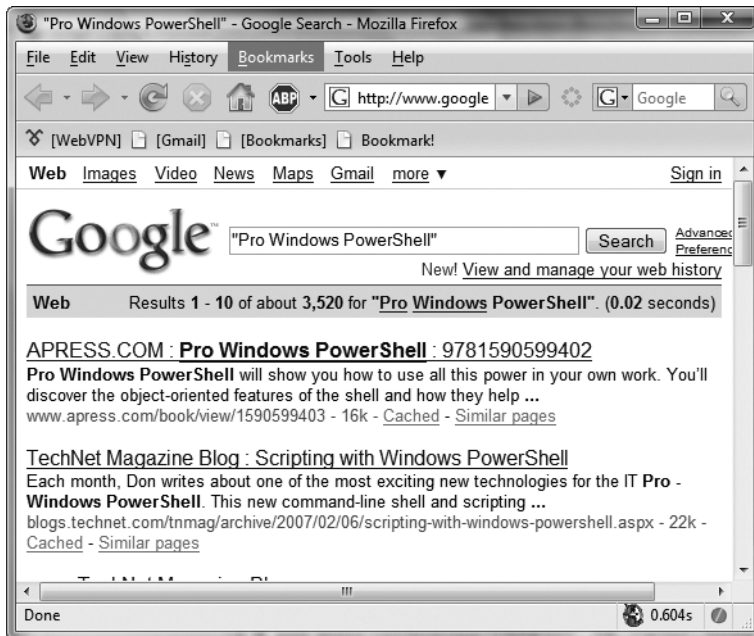


Figure 13-4. A Google search triggered from within PowerShell

Here is our implementation of the Search-LiveCom function that triggers a Live.com search:

```
function Search-LiveCom($query)
{
    $encodedQuery = Encode-Url $query
    Open-Url "http://search.live.com/results.aspx?q=$encodedQuery"
}
```

Let's try the same search that we did with Google. The result is shown in Figure 13-5.

```
PS> Search-LiveCom '"Pro Windows PowerShell"'
PS>
```

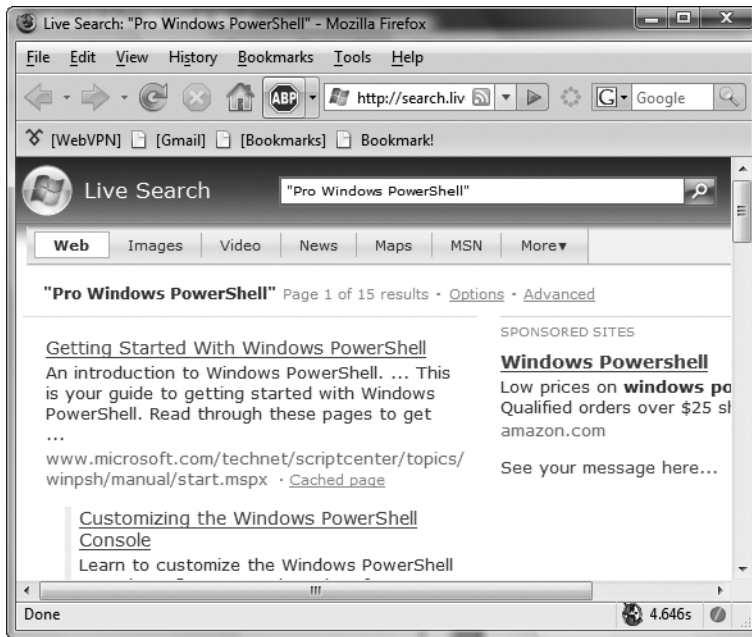


Figure 13-5. A Live.com search triggered from within PowerShell

You can find the source code for the Internet-related functions in the `Help-SearchWeb.ps1` file. It is best to copy and paste them in your shell profile script, so that the commands are available all the time. For information about modifying your profile configuration, refer to Chapter 11.

Summary

Gaining access to information in all situations is an invaluable skill and an important tool in your problem-solving arsenal. This chapter showed how to use the help system to get various bits of information about the available commands. It then described how to discover new commands and how to examine live objects and get information about their properties and methods.

At the end, I showed you how to use the biggest resource of all—the Internet—to get help. Using the approach outlined in the `Search-*` functions, you can go on to create functions that search for answers and code in online forums and script repositories. After all, why bang your head against a problem when you have the entire Internet at your disposal? Paraphrasing the old saying, we can go by the maxim, “Good scripters know how to write good code; excellent scripters use code that has already been written before.”



Taming Processes and Services

Processes are like living organisms that inhabit a running operating system. They are the entities that do all the work—they munch on all the data and pass it to each other. While much work can be done from within PowerShell only, sooner or later we will have to interact with one of the processes on our system. All shells provide facilities for managing running programs and so does PowerShell. Its process and service management facilities can extract information about processes and manipulate process instances by working directly with objects that represent them. We have convenient, object-oriented commands that give us great flexibility when extracting data or modifying state. The combination of the objects representing processes and services with the object pipeline and that pipeline's advanced iteration and filtering facilities makes expressing complex commands easy.

In this chapter, you will learn how to work with processes and their properties. We will start and stop processes, query them for information, and use them to display statistics. We will then continue with administering system services. Traditionally, working with processes and services has been done either manually from the Windows GUI or from complex WMI scripts written in VBScript or JScript. There is a third option: nonprogrammers can call command line utilities such as `tasklist.exe`, `taskkill.exe`, or `sc.exe` from batch files to start or stop processes and services. PowerShell can and will make those three options obsolete by providing a standard, easy-to-use interface to performing these tasks. Apart from the excellent WMI support that we will be discussing in Chapter 20, PowerShell provides replacement cmdlets for the WMI-based solutions that are easier to use and will save us a lot of time.

Working with Processes

All operations related to processes in PowerShell are supported by two cmdlets: `Get-Process` and `Stop-Process`. Here is how to verify that by getting all cmdlets that relate to the `Process` noun:

```
PS> Get-Command -Noun Process
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Get-Process	Get-Process [-Name] <S...
Cmdlet	Stop-Process	Stop-Process [-Id] <Int...

Listing Processes: Getting the Big Picture

Sometimes, our system becomes unresponsive for no apparent reason. The experienced user usually fires up a task manager instance and starts poking around. Is there a process eating too much CPU time? Do we have a program that has gone wild and has eaten up all available memory on our system and is that what is causing the operating system to thrash? The task manager is a great tool, but it is not very flexible. It just gets us a list of processes and allows us to sort them on a column like CPU, memory, and so on. PowerShell allows us to quickly craft various reports about the system using the `Get-Process` cmdlet and a couple of pipeline tricks. For starters, we can sort all processes on a property value and display the top offenders. This is how we get the top five virtual memory users:

```
PS C:\> Get-Process | sort VM -descending | select -first 5
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
381	10	32224	1132	1505		1932	sqlservr
267	13	121888	114048	262	12.23	2400	Reflector
421	18	32196	25148	255	10.42	2804	devenv
723	6	45700	43232	245	39.06	3176	javaw
845	41	54884	51696	245	183.05	2680	explorer

Seeing a lot of virtual memory used by a program does not necessarily mean that exactly that program is causing the system slowdown. To get a more accurate picture, we may need to examine the working sets by sorting on the `WS` property:

```
PS C:\> Get-Process | sort WS -descending | select -first 5
```

If we suspect thrashing, we can look at the paged and nonpaged memory sizes using the `PM` and `NPM` properties, respectively.

Note Thrashing is a term that is used to describe the situation where all the physical memory has been used up, and the system uses virtual memory for further memory allocations. When that reaches a certain point, the system is kept busy swapping memory pages to disk and to memory, and the system slows almost to a halt.

We can get all processes that occupy more than 50MB of physical memory. Those types of operations are so common that PowerShell supports suffixes like KB, MB, GB for number literals. Here is our query:

```
Get-Process | where { $_.WS -ge 50MB }
```


Starting and Stopping Processes

All process-related operations in PowerShell involve working with a `.NET System.Diagnostics.Process` object. Starting a process is easy—we just have to type the executable name. But how do we get the `Process` object instance for that process and interact with it later? One possible approach is to use `Get-Process` right after starting it:

```
PS C:\> notepad
PS C:\> Get-Process notepad
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
54	3	1864	7324	42	0.09	3892	notepad

What if we have more than one Notepad instance running? We can sort them on process ID, or even better, on the `StartTime` property. This still does not protect us from the hypothetical situation where a user starts a Notepad instance after we start our instance and before our `Get-Process` cmdlet executes, so this is not a good way to get the just-started process. The safest way is to start the process using the `Process.Start()` static method, as it will return the newly created process. Here is how to do it:

```
PS C:\> $notepad = [Diagnostics.Process]::Start("notepad.exe")
PS C:\> $notepad
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
51	3	1860	7188	42	0.13	2816	notepad

Tip The PowerShell Community Extensions (PSCX) project solves this problem in the best way possible. It provides a `Start-Process` cmdlet that we can use to start a new process and get a reference to it, at the same time. Please refer to Chapter 21 for additional information on obtaining and installing PSCX.

Stopping a process is easy if we do not care if the process will lose any data. Let me paraphrase that—killing a process is easy. There are two ways to do it. You can call the `Kill` method on the process:

```
$notepad.Kill()
```

or pass process instances to the `Stop-Process` cmdlet, which is appropriately aliased to `kill`:

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

Ending a process gracefully, without it losing data, is a lot harder. We can attempt to do so by calling the `CloseMainWindow` method—that will notify the process, and there is good chance that it will exit normally. If it does not, we can still terminate it. Here is a possible solution that tries to close a Notepad process, waits two seconds, and then terminates it if it is still running:

```

$notepad = Get-Process notepad

echo "Trying to close the process window..."
$messageSent = $notepad.CloseMainWindow()
sleep 2
if (!$notepad.HasExited)
{
    echo "Forcing process termination."
    Stop-Process -input $notepad
}
else
{
    echo "Process exited gracefully"
}

```

Unfortunately, `CloseMainWindow` only returns a value that signals if the method has managed to send the “Close window” message to the target process. We still have to check if the process has honored that message and has really exited. We can check that by inspecting the `HasExited` property.

Processes and Their Windows

We cannot interact directly with Windows and pass messages around. Well, it may be possible if we script objects from the `System.Windows.Forms.NET` namespace, but doing that will not be easy. The `Process` class exposes a convenient property, `MainWindowTitle`, that we can use to distinguish among several running instances of the same program. Suppose we need to find the browser that displays the Google search page. Here is a solution that uses a wildcard search on the window title:

```
Get-Process iexplore | where { $_.MainWindowTitle -like "*google*" }
```

Usually, window titles contain document titles, file names, and other important data. That makes filtering on the `MainWindowTitle` property a really powerful tool in identifying the right process.

Another useful GUI-related `Process` property is `Responding`, which tells us if the main window thread is processing messages or, in other words, if the application responds to user input. We can use this property to get all hung processes and, well, kill them. We do not want to brutally murder all nonresponding processes, as there is a good chance that some of them may be doing a CPU-intensive task and will recover any moment. A seemingly good solution is to get the hung processes, wait for several seconds, and kill those that are still stuck in limbo:

```

$before = (Get-Process | where { $_.Responding -eq $false })
sleep 5
$after = (Get-Process | where { $_.Responding -eq $false })

```

```
if (($before -ne $null) -and ($after -ne $null))
{
    diff $before $after -includeEqual |
        where { $_.SideIndicator -eq "==" } |
        foreach { $_.InputObject | Stop-Process }
}
```

We get the hung processes, wait five seconds, and get them again. We then compare the two results and kill all processes that are present in both collections. Note the `-includeEqual` `diff` option and the comparison that verifies that a `SideIndicator` has a value of `==`. That means that the object is present in both collections or the process has failed to recover in those five seconds.

Process Modules: On What Libraries Does This Baby Depend?

Process modules are a fancy name for all the DLL and EXE files that are loaded in a process's memory space. We have two properties on the `Process` class that help us work with modules: `MainModule` and `Modules`. The main module is usually the EXE file that started the process. The `Modules` collection contains all dynamic libraries that have been loaded.

`MainModule` is a very convenient property, a feature that has been missing in the classic `cmd.exe` shell. We can use `MainModule` to get the path to the executable file. Have you ever tried looking for `PowerShell.exe` somewhere in your `Program Files` folder? I have—before finding out that it is buried deep below the `Windows` folder. Here is how you can find out for yourself where that file lives:

```
PS C:\> (Get-Process powershell).MainModule.FileName
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

You can use the `Modules` collection to find out on what libraries a process depends. Here is the start of the list for the `Windows` calculator:

```
PS C:\> (Get-Process calc).Modules
```

Size(K)	ModuleName	FileName
184	calc.exe	C:\Windows\sys...
1144	ntdll.dll	C:\Windows\sys...
...		

Knowing what a specific library does gives us greater insight into how a program works. We can even generate various reports that find similar programs. For example, we know that all processes that use `mscorlib.dll` are .NET processes. Here is how to get them:

```
Get-Process |
foreach { $proc = $_; $proc.Modules |
    where { $_.ModuleName -eq "mscorlib.dll" } |
    foreach { echo $proc }
}
```

This is some monster pipeline, and it needs some explanation. The first `foreach` call iterates over all processes and gets their modules. We save the current process object in a temporary variable called `$proc`. We then filter the modules for the process using the `where` cmdlet to get only modules that are actually `mscorlib.dll`. If we find any, that will be exactly one module. We pipe that module through the inner `foreach` call—we rely on the fact that the `foreach` script block is executed only if the `where` command has returned a matching module. We do not really care about the module—we just echo the previously saved process. Here is a sample result featuring Visual Studio .NET (`devenv.exe`), PowerShell, and Reflector (the .NET code decompiler):

```
PS C:\> .\DotNetProcesses.ps1
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
421	18	32624	25128	255	12.05	2804	devenv
758	7	56368	30804	175	34.52	3956	powershell
267	13	121888	97336	262	12.23	2400	Reflector

Getting Information About a Program’s Publisher

Have you ever found a process with a suspicious name and wondered if it is not the latest spyware program or virus? It is always best to run a regularly updated antivirus solution, but just for your research purposes, you can examine the process properties that return the company, product, and product version. Again, those can be spoofed by a malicious person, and any data should be taken with a grain of salt. Here is a portion of a tabular report I ran for my processes today:

```
PS C:\> Get-Process | select Company,Product,ProductVersion | format-table
```

Company	Product	ProductVersion
-----	-----	-----
Microsoft Corporation	Microsoft® Visual Studi...	8.0.50727.762
Sun Microsystems, Inc.	Java(TM) 2 Platform Sta...	5.0.120.4
Microsoft Corporation	Windows Defender	1.1.1505.0
Don HO don.h@free.fr	Notepad++	4.1.2

Setting Process Priority

Raising and lowering process priorities is a task that I do quite often. Installing a huge program takes time, and I would like to do other work while the installer copies files around. I usually lower the installer process priority, so that the rest of the applications run smoothly. I raise the priority back to normal if I am done with other work or just want the installer to finish sooner.

Setting a process’s priority is done by setting the `PriorityClass` property. It is an enumeration property, so in PowerShell you have to pass a string like “`BelowNormal`”, “`Normal`”, and so on. Here is how we can lower an installer’s (`msiexec.exe`) priority and get it back to normal if we wish to:

```

echo "Lowering installer process priorities"
Get-Process msixec -ErrorAction Stop |
    foreach { $_.PriorityClass = "BelowNormal" }
Read-Host -prompt "Press Enter to restore installer priorities"
Get-Process msixec -ErrorAction SilentlyContinue |
    foreach { $_.PriorityClass = "Normal" }

```

The script will restore the priority to “Normal” after the user presses the Enter key. Note the `ErrorAction` parameters: initially, we want the script to fail if there are no `msiexec.exe` processes running, and we ignore any errors after the wait as the installation may have finished already. Note that installing software requires administrator privileges, so make sure you run the script under an administrator account.

Administering Services

When working with services, PowerShell passes around `System.ServiceProcess.ServiceController` objects. The simplest way to get those objects for one or many services is to use the `Get-Service` cmdlet. When called without parameters, it will return all services registered on the system. We can use the cmdlet to generate quick reports. For example, this command will list just the running services:

```

Get-Service | where { $_.Status -eq "Running" }

```

Here is what the result looks like:

Status	Name	DisplayName
Running	AeLookupSvc	Application Experience
Running	Appinfo	Application Information
Running	AudioEndpointBu...	Windows Audio Endpoint Builder
Running	Audiosrv	Windows Audio
...		

We can search for services by name using wildcards. That is very useful in cases where you do not remember the exact name of a service. Here is how to get the DNS Client service by getting all services whose names are like the `DNS*` wildcard:

```
PS> Get-Service DNS*
```

Status	Name	DisplayName
Running	Dnscache	DNS Client

Another good use for wildcards in service names is when you want to find related services, as they usually have similar names. Here is how to get all services related to Microsoft SQL Server by getting all instances that match the `*SQL*` wildcard:

```
PS C:\> Get-Service *SQL*
```

Status	Name	DisplayName
-----	----	-----
Running	MSSQL\$SQLEXPRESS	SQL Server (SQLEXPRESS)
Stopped	MSSQLServerADHe...	SQL Server Active Directory Helper
Stopped	SQLBrowser	SQL Server Browser
Running	SQLWriter	SQL Server VSS Writer

Caution Wildcards get applied to the service name, not the display name. That is why a wildcard, such as `SQL*`, would not get the main SQL Server service instance—it is called `MSSQL$SQLEXPRESS`. The same holds true for the DNS Client service example; note that the actual service name is `Dnscache`.

Starting and Stopping Services

Windows already provides a console tool that can start and stop services, `net.exe`, but it is not the best tool for the job. That program is complex, and it already tries to do too much. For example, it can even add and remove shared drives. That is why PowerShell provides built-in cmdlets for starting, stopping, pausing, and resuming services, all of which are pretty straightforward to use. Starting a service is done with `Start-Service`:

```
Start-Service W3SVC
```

You can do the same by calling the `ServiceController` `Start` method:

```
(Get-Service W3SVC).Start()
```

Note that changing service status requires administrator privileges. Stopping a service can be done with the `Stop-Service` cmdlet:

```
Stop-Service W3SVC
```

or the `Stop` method:

```
(Get-Service W3SVC).Stop()
```

Whether you use the cmdlet or object method is purely a matter of preference and convenience. The rest of the service status manipulation cmdlets—`Restart-Service`, `Suspend-Service`, and `Resume-Service`—work in absolutely the same way.

Caution Starting, stopping, suspending, and resuming services are all operations that may impact the system in a serious way, and those operations require administrator privileges. If running on Windows Vista, make sure that you run the preceding examples in a shell session running with elevated privileges.

Configuring Services

PowerShell provides a cmdlet, `Set-Service`, that allows you to change almost anything about a service. You can even change the service name and description. I consider this dangerous, because you run the risk of not being able to find your service a week after renaming it; you may have valid reasons for doing so. For me, the most important feature of `Set-Service` is the ability to change a service's startup type. Here is how we can prevent the `W3SVC` service from being started by changing its startup type to `Disabled`:

```
Set-Service W3SVC -startupType Disabled
```

Making the same service run on every startup is a matter of changing the startup type back to `Automatic`:

```
Set-Service W3SVC -startupType Automatic
```

A word of caution—we have to be very careful when starting and stopping services and changing their startup types. Some services are critical for Windows operation and need to be running all the time; stopping or disabling them may render your system unusable.

Analyzing Service Dependencies

Services often depend on other services. The Services management console snap-in will display those dependencies, but sometimes, we may need that information from a PowerShell session or a script. To get that data, we have to read the `ServiceController` `DependentServices` and `ServicesDependedOn` properties. The former returns services that depend on the current one, and the latter gets us the services that the current service depends on. Let's get this information for the WMI service, `winmgmt`:

```
PS C:\> (Get-Service winmgmt).DependentServices
```

Status	Name	DisplayName
-----	----	-----
Running	wscsvc	Security Center
Stopped	SharedAccess	Internet Connection Sharing (ICS)
Running	iphlpvc	IP Helper

```
PS C:\> (Get-Service winmgmt).ServicesDependedOn
```

Status	Name	DisplayName
-----	----	-----
Running	RPCSS	Remote Procedure Call (RPC)

As you can see, the WMI service depends on the Remote Procedure Call service, while the Security Center, Internet Connection Sharing, and IP Helper services depend on it.

Summary

Most, if not all, of the operations presented in this chapter can be performed using a mixture of preexisting Windows utilities and/or WMI scripts. PowerShell does nothing revolutionary here. Its goal is to unify the access to processes and services and creates a homogeneous environment that makes scripting a breeze. The cmdlets presented here are easy to learn and master, and that is the biggest step forward in system administration and automation.



Input and Output

Data comes in all flavors and formats. It would be great if all systems were using objects, and PowerShell could use those objects to get data from any system. That is a good thing to strive for, but unfortunately, it is not the case at all. The good old file continues to reign as the primary means for communication between heterogeneous systems. All-powerful file formats come and go, but the only thing that can really get read and written by just about any computer system on this planet is plain text. We do not need to run a program written for an IBM mainframe ten years ago on our workstation as long as we can get some of the data that program generates in text format. Of course, text files will surely have different formats, but the best thing about a computer-generated text file is that it is generally easy to parse using a computer.

Data generated by programs tends to outlive the programs that created it in the first place. That is what makes text files attractive—they can be easily transferred from one system to another and live on after their original creator has long been forgotten. In this chapter, we will delve into PowerShell's facilities for working with files. We will be reading different data formats and generating some data ourselves. I will then show you how to use regular expressions to get the actual data from the text blob. Let's get started!

Reading Content

Everything about a file's content in PowerShell revolves around the two cmdlets `Get-Content` and `Set-Content`. While they can get and set raw binary data, by default, they are oriented toward working with text. For example, here is how we can use `Get-Content` to retrieve the text stored in a sample file called `test_people.txt`:

```
PS> Get-Content .\test_people.txt
Michael Johnson
John Smith
```

```
Jane Doe
```

Users with a background in a different programming language may expect that `Get-Content` will return a string with the file contents. Not quite—the cmdlet is line oriented and returns an array of strings, each representing a line from the original text. Here is how to verify that; the result returned from `Get-Content` and our file is an array of four strings:

```

PS> $content = Get-Content .\test_people.txt
PS> $content.Length
4
PS> $content[0]
Michael Johnson
PS> $content[0..3]
Michael Johnson
John Smith

Jane Doe

```

As you can see from the preceding code, we have everything already processed for us, and we can just go over the lines one at a time to get people's names. That is pretty convenient when working with files containing lists of items stored one item per line.

Convenient as it may be sometimes, working with line-oriented operations at other times may be more of a nuisance than a helper. We may need to get the entire text file contents as a single string, so that we can process it in some other way that is not line oriented. We can do that by using the static `[string]::Join()` method that takes an array of strings and joins them into a single string using a delimiter value. Our delimiter value should be the new line symbol—but which one? Different systems use different symbols to indicate a line ending: Windows uses carriage return and line feed (``r`n` in PowerShell escape symbols); UNIX systems use just a line feed (``r`); while Macintosh machines use just a carriage return (``n`). It is true that PowerShell currently runs on Windows only, and we can use ``r`n` as our new line separator, but it is best to get the new line symbol from our environment, so that we future-proof our script in case someday PowerShell runs on one of those systems. To get the new line symbol from the .NET environment, we need to access the static `[System.Environment]::NewLine` property. Using that information, we can create a small utility script, called `Get-ContentAsString.ps1`, which can read a file and return its entire contents as a single string. Here is the code:

```

param ($Path)

$lines = Get-Content -Path $Path
$newLine = [System.Environment]::NewLine

$content = [string]::Join($newLine, $lines)
$content

```

Here is the result when we run the script on our `test_people.txt` file and examine the object that we get as a return value:

```

PS> $content = .\Get-ContentAsString.ps1 .\test_people.txt
PS> $content.Length
39
PS> $content[0..6]
M
i
c
h
a
e
l

```

We get a string containing 39 characters, and the first 7 of those are the characters representing the string “Michael”.

One of my favorite features of `Get-Content`’s is that it can target several files. We can pipe files to it, and it will return their contents. The result is a single collection of lines containing the combined content of all files. Here is how we can get the contents of all `test*.txt` files in the current folder. There are two of them: `test_people.txt`, which you already saw, and `test_numbers.txt`, which contains several numbers, each on a separate line:

```
PS> dir test*.txt | Get-Content
```

```
1
2
3
4
5
Michael Johnson
John Smith
```

```
Jane Doe
```

We can achieve the same result by passing a wildcard as the file path parameter straight to `Get-Content`. Here is how to do it:

```
PS> Get-Content test*.txt
```

```
1
2
3
4
5
Michael Johnson
John Smith
```

```
Jane Doe
```

Our `Get-ContentAsString.ps1` is a really thin wrapper over `Get-Content`’s functionality, and the fact that we pass the `$Path` parameter straight to the cmdlet allows it to work with wildcards too:

```
PS> .\Get-ContentAsString.ps1 test*.txt
```

```
1
2
3
4
5
Michael Johnson
John Smith
```

```
Jane Doe
```

Although this output looks the same as the previous example, it is structurally different. The example before returns a collection of strings, and this one returns a single string.

Writing Content

As we already mentioned, `Get-Content` has a symmetric cmdlet that performs the opposite action. It is called `Set-Content`, and it takes two parameters: `-Value`, which contains the object to be written to the file, and `-Path`, which holds the path to the file. The `-Value` parameter can be taken from the input pipeline instead of being specified explicitly. I have mentioned that it contains the object to be written to the file; this can be any object or a collection. If you pass an object, its built-in `ToString()` method will be called to get the string representation that will be stored to the file. If you pass a collection, it will be enumerated and all its members will be treated in manner that applies for a single object. To see that in action, let's write two files to a text file:

```
PS> $files = dir test*.txt
PS> Set-Content -Value $files -Path texts.txt
PS> Get-Content .\texts.txt
Z:\15 - Input and Output\test_numbers.txt
Z:\15 - Input and Output\test_people.txt
```

You may be guessing what is going on already. The `FileInfo` objects that we get from the `dir` command have their `ToString()` method return the full path to the file.

You should be aware that the `Set-Content` cmdlet is destructive by nature. If the file that we are trying to write to already exists, its entire contents will be overwritten with the new ones. Be careful when specifying your paths, because `Set-Content`'s destructive nature may cause you to lose data.

On many occasions, we want to append some content to an existing file, rather than overwrite it. PowerShell has its `Add-Content` cmdlet for those cases. It takes exactly the same parameters as `Set-Content`, but it will not destroy any content. Here is how we can use the `Add-Content` cmdlet to append another file name to our `texts.txt` file:

```
PS> dir Get*.ps1 | Add-Content .\texts.txt
PS> Get-Content .\texts.txt
Z:\15 - Input and Output\test_numbers.txt
Z:\15 - Input and Output\test_people.txt
Z:\15 - Input and Output\Get-ContentAsString.ps1
```

This time, I decided to pipe the `dir` command result straight to `Add-Content`. The command works in absolutely the same way as if the file list has been provided through the `-Value` parameter.

PowerShell supports two output redirection operators that allow us to save a command's output to a file. The `>` operator will send the current object to a file overwriting its original contents. Here is how it works:

```
PS> dir Get*.ps1 > texts.txt
PS> Get-Content .\texts.txt
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::Z:\15 - Input and
Output
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/10/2007 12:25 AM	148	Get-ContentAsString.ps1

Note that we have lost any of the original content. The >> operator behaves in a manner similar to > with the only difference being that it does not destroy a file's previous contents but appends the data at the end instead. Here is how we can use it to append another file description to our texts.txt file:

```
PS> Get-Content .\texts.txt
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::Z:\15 - Input and
Output
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/10/2007 12:25 AM	148	Get-ContentAsString.ps1

```
Directory: Microsoft.PowerShell.Core\FileSystem::Z:\15 - Input and
Output
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/10/2007 12:21 AM	39	test_people.txt

You can see that the > and >> operators work very similar to Set-Content and Add-Content. An important difference to note is that the redirection operators pipe the input objects through PowerShell's formatting facilities, instead of calling their ToString() methods as Set-Content and Add-Content do. You can find detailed information about PowerShell's object formatting infrastructure in Chapter 12.

Content Encoding

Text is an interesting thing. Computers were invented by people who spoke English, and the first text formats, like the American Standard Code for Information Interchange (ASCII), used only a single byte to represent a character. This put a limit on the number of characters that can be represented. The encoding is suitable to represent the Latin alphabet, but fails completely for other character sets like Cyrillic, Hebrew, Arabic, Chinese, Japanese, Korean, and so on. That is why the Unicode Consortium folks have come up with a standard way of encoding text that should be able to accommodate all character sets.

The PowerShell cmdlets like Get-Content and Set-Content support specifying the encoding that should be used when reading from or writing to the file through the -Encoding parameter. Here are the possible values:

- **Unknown:** No encoding is used. The content is treated like a normal Unicode string.
- **String:** This treats content as a Unicode string; it's the same as Unicode.
- **Unicode:** The default string format in the .NET framework and PowerShell represents a Little Endian 16-bit Unicode encoding. See the section about the different Unicode encodings later in this chapter for details on Endianness and byte order.
- **Byte:** This will return an array of bytes and is suitable for binary manipulation of a file.
- **BigEndianUnicode:** This value is similar to Unicode but uses Big Endian byte order.
- **UTF8:** This uses the Unicode Consortium standard UTF-8 encoding.
- **UTF7:** This uses the Unicode Consortium standard UTF-7 encoding.
- **Ascii:** This returns file contents in the ASCII encoding that uses 8 bits per character and is suitable for English text only.

Of the possible values in this list, the most interesting are **Byte**, **Unicode**, **UTF8**, and **Ascii**, as they will be the ones you are most likely to use.

Getting Byte Content

The most powerful way to work with a file's contents is to get it as a byte array. That grants you the ability to change every single bit in the file and turn it into whatever you wish. Let's now save a file as an ASCII character sequence and get its byte values:

```
PS> "aaaa" | Set-Content -Encoding Ascii -Path ascii.txt
PS> Get-Content .\ascii.txt -Encoding Byte
97
97
97
97
13
10
```

We are using the sample sequence of “aaaa”, so that we can easily distinguish it in the byte sequence we get back. As you can see from the results, our file contains 6 bytes: the four “a” characters plus two characters with the 10 and 13 character codes, respectively. The 10 code corresponds to the carriage return symbol, and the 13 to the line feed one. We get those two because **Set-Content** automatically appends a new line at the end of the file.

Caution Be careful with the new line that **Set-Content** automatically inserts for you. For text files, the line may not be a problem, but the new line can break the file if you need to set the content specifically without new lines at the end. The only way to set a file's content without the cmdlet is to use **Byte** encoding and provide a byte array with the contents.

Working with bytes is often easily done when displaying byte values in hexadecimal. In addition, all documentation about binary data uses hexadecimal. Let's create a short script that will help us in formatting the numbers in hex. We will call it `Format-AsHex.ps1`, and it will use the `-f` string format operator to format all numbers that have been piped in hex. Here is the code:

```
process
{
    "{0:X}" -f $_
}
```

Let's now use it for our ASCII data:

```
PS> Get-Content .\ascii.txt -Encoding Byte | .\Format-AsHex.ps1
61
61
61
61
D
A
```

As you can see, the hexadecimal code for the “a” character is 61, the carriage return symbol is D, and the line feed one is A.

Different Unicode Encodings

Now that we have the tools for looking at binary data, we can see how text is saved under the different encodings. Let's start with the .NET-standard Unicode encoding. We will save our sequence of characters in that encoding and get the file bytes:

```
PS> "aaaa" | Set-Content unicode.txt -Encoding Unicode
PS> Get-Content .\unicode.txt -encoding Byte | .\Format-AsHex.ps1
FF
FE
61
0
61
0
61
0
61
0
D
0
A
0
```

According to the standard, Unicode text files should start with a sequence of bytes that identify the byte order; they are called the byte order mark, or BOM. The BOM is necessary because some machines represent 2-byte numbers, like the code for the “A” character in the

0061 form, similar to the normal mathematical notation. Others invert the byte order and would represent the same character as 6100. The former ordering is called Big Endian, and the latter is Little Endian. Intel-based machines and the .NET Framework use Little Endian ordering. The BOM at the beginning of the file contains the number FFFE; according to the standard, that indicates Little Endian encoding. You can see that the “A” characters are encoded as 6100. .NET calls that encoding Unicode, and its inverse, BigEndianUnicode.

Let’s now save the same sequence into BigEndianUnicode encoding and take a look at the bytes:

```
PS> "aaaa" | Set-Content unicodeBE.txt -Encoding BigEndianUnicode
PS> Get-Content .\unicodeBE.txt -encoding Byte | .\Format-AsHex.ps1
FE
FF
0
61
0
61
0
61
0
61
0
D
0
A
```

The BOM now contains FEFF, and as you would expect, “A” characters are represented by 0061.

Both the Unicode and BigEndianUnicode encodings use two bytes per character, and that makes them a bit wasteful. English characters usually occupy only 1 byte, and the second one is always zero. The UTF-8 encoding solves that problem by using 1 byte for English characters and 2 bytes or more for other characters. Let’s now save our letter sequence in UTF8 and take a peek at the bytes:

```
PS> "aaaa" | Set-Content utf8.txt -Encoding UTF8
PS> Get-Content .\utf8.txt -encoding Byte | .\Format-AsHex.ps1
EF
BB
BF
61
61
61
61
D
A
```

The data is a lot more compact now, but what are those three bytes before our four 61 bytes that represent the “A” characters? That triplet is the so-called UTF-8 BOM sequence. The UTF-8

format does not suffer from byte order issues, but it can specify a BOM sequence. Programs usually use it to detect if a text file is encoded using UTF-8.

Note You may have noticed these three strange symbols on web pages or inside text files: `ï»¿`. They are really the UTF-8 BOM symbols. The EFBBBF byte sequence has a printable representation, and programs that do not know how to handle UTF-8 usually display those characters to the user.

Now that we know about the different BOM sequences, we can use that knowledge to write a script that will detect the encoding a text file uses. We will call it `Detect-Encoding.ps1`, and it will read the first 3 bytes from a file and compare them against the known BOM values. If it finds a match, it will print the corresponding encoding name. If it does not, it will assume it is dealing with an ASCII file. Here is the code:

```
param ($Path)

$start = Get-Content -Path $Path -Encoding Byte -TotalCount 3

$utf8BOM = "{0:X}{1:X}{2:X}" -f $start
$utf16BOM = "{0:X}{1:X}" -f $start

if ($utf8BOM -eq "EFBBBF")
{
    Write-Host "UTF-8"
    exit
}

if ($utf16BOM -eq "FFFE")
{
    Write-Host "Unicode"
    exit
}

if ($utf16BOM -eq "FEFF")
{
    Write-Host "Big Endian Unicode"
    exit
}

Write-Host "No BOM detected. Encoding is most likely ASCII."
```

The code passes the `-TotalCount` parameter to `Get-Content` in order to get just the first 3 bytes. It formats the 3 bytes as a hex sequence and tests if that sequence is equal to the UTF8 BOM. If not, it formats the first 2 bytes and compares them to the Unicode and UnicodeBigEndian BOM sequences. Here is how it works for the test files that we have already generated:

```
PS> .\Detect-Encoding.ps1 .\ascii.txt
No BOM detected. Encoding is most likely ASCII.
PS> .\Detect-Encoding.ps1 .\unicode.txt
Unicode
PS> .\Detect-Encoding.ps1 .\unicodeBE.txt
Big Endian Unicode
PS> .\Detect-Encoding.ps1 .\utf8.txt
UTF-8
```

Note The home site of the Unicode Consortium is located at <http://www.unicode.org/>. The site hosts the entire Unicode standard specification. The standard is the definitive reference on the subject, but it may get a bit lengthy and hard to read. For an excellent introduction on the subject of text encodings, you can read Joel Spolsky's "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)" article available at <http://www.joelonsoftware.com/articles/Unicode.html>. Do not get intimidated by the article's subject—it really does not require any programming experience to get the concepts.

Extracting Data from Text

The text data that we get our hands on is rarely in the format we need. As a contrived example, take the string in our `test_numbers.txt` file. How do we extract the numbers from a string that contains numbers separated by whitespace and other separators? How do we locate the correct position in the string and extract the digit portion only? The answer is regular expressions.

Note For more information on regular expressions, you can visit the excellent online resource that is the <http://www.regular-expressions.info> site and go through its tutorials and references on the subject. The site covers regular expression usage in different programming languages and environments. While the syntax and support is more or less the same on most platforms, the regular expressions syntax still has some subtle differences, so make sure you go through the article's subject about .NET regular expressions at <http://www.regular-expressions.info/dotnet.html>. PowerShell uses the .NET regular expressions internally.

Finding Matches for a Regular Expression

Getting back to our problem, we need to create a regular expression that will describe a number. We can do it by specifying `\d+`, which means "one or more digit symbols." We then need to apply that expression over the string and get all the matches. We can do that by calling the regular expression's `Matches()` method. It returns a collection of `Match` objects. We need to check if every match is successful by verifying its `Success` property. If a match is successful, the value of the string that matched the regular expression, our much-needed number, is available

through the `Value` property of the first `Group` object. To solve the problem, we will create a new script, `Extract-Numbers.ps1`. Here is its code:

```
$nums = .\Get-ContentAsString.ps1 .\test_numbers.txt
$numberMatcher = [regex] "\d+"

$matches = $numberMatcher.Matches($nums)
foreach ($match in $matches)
{
    if ($match.Success)
    {
        $number = $match.Groups[0].Value
        Write-Host "number: $number"
    }
}
```

Note that the first group is available at index 0. Here is the script output:

```
PS> .\Extract-Numbers.ps1
number: 1
number: 2
number: 3
number: 4
number: 5
```

Finding String Occurrences Inside Files

Seasoned shell scripters are, without doubt, aware of utilities that search for a given string or regular expression pattern's occurrences in one or many files. The typical UNIX utility that does that is `grep`, and on Windows, this job is done by the `findstr` tool. PowerShell can use those utilities if they are available, but we are better off using the built-in `Select-String` cmdlet that offers the same functionality.

The most important `Select-String` parameters are `-Pattern` and `-Path`. The former contains the string or regular expression that we are looking for. The latter contains the file name or wildcard that specifies the files we will be searching in. To see the cmdlet in action, we can use it to find all occurrences of the string "if" in all PowerShell scripts in the current folder:

```
PS> select-string if *.ps1

Detect-Encoding.ps1:8:if ($utf8BOM -eq "EFBBBF")
Detect-Encoding.ps1:14:if ($utf16BOM -eq "FFFE")
Detect-Encoding.ps1:20:if ($utf16BOM -eq "FEFF")
Extract-Numbers.ps1:7:    if ($match.Success)
```

Of course, `Select-String` is tightly integrated with PowerShell's object-oriented infrastructure. The result we get is not just a bunch of text lines; it is a collection of `MatchInfo` objects. Those objects contain information about the file and the line where the match was found. Here is how to get that information using the `select` cmdlet to retrieve the `Line`, `FileName`, and `Path` properties:

```
PS> select-string if *.ps1 | select Line,FileName,Path
```

Line	Filename	Path
----	-----	----
if (\$utf8BOM -eq "EF... Detect-Encoding.ps1		Z:\15 - Input and O...
if (\$utf16BOM -eq "F... Detect-Encoding.ps1		Z:\15 - Input and O...
if (\$utf16BOM -eq "F... Detect-Encoding.ps1		Z:\15 - Input and O...
if (\$match.Success) Extract-Numbers.ps1		Z:\15 - Input and O...

Regular expressions are a very powerful tool for working with all kinds of text, though they are a bit hard to learn and may require a larger portion of your time. Trust me—investing the time will pay off in the future. Once you master regular expressions, you will be able to extract data from almost any program.

Summary

In this chapter, you learned how to work with file contents: how to retrieve a file, process it, and save it back to the hard drive. We did a quick tour of the most popular text encodings that may come in handy if you have to process a file originating from a little-known system. At the end, I showed you how to find and extract data from freeform text using regular expressions. With these tools in your toolbox, you should be able to process text coming from different programs and extract the data you need.



Monitoring Your System

Computers running on a network can sometimes be likened to living organisms. A sophisticated network is like a complex ecosystem; it has all sorts of organisms, each with its own specific capabilities and requirements. Our PowerShell scripts and commands issued at the interactive shell are usually in the position of instructing other system components on what to do and how to do it. Being proactive in anticipating possible problems and failures is one of the characteristics a successful system administration strategy must have. Unfortunately, it is not always possible to anticipate and prevent every single problem. In Chapter 9, you learned how to trap and handle errors generated from within scripts. In this chapter, we will be working with errors triggered by other programs. You will learn important techniques for monitoring a system and its components and watching their operation. Our goal is to spot each anomaly and failure as soon as it happens. In most cases, we can even react and fix the problem before regular users even notice it! Even if we cannot fix things that quickly, it is always better to know in advance and notify interested parties, so that the problem has minimal impact on people's work processes.

In this chapter, we will use two tools that are the pillars of system monitoring: the event log and performance counters. The event log is a central location that is used both by user programs and the operating system to store information messages; you will learn how to read event log entries from within scripts and how to create new entries. Performance counters enable application and operating system components to publish performance-related data and can be used to gather a lot of interesting types of performance statistics. While data gathering may be interesting enough in itself, we are most interested in taking action, so our performance analysis will be primarily targeted at taking action whenever our monitored application reaches some performance threshold.

Working with the Windows Event Log

The Windows event log is a central location that is aimed at collecting all sorts of diagnostic messages. On many occasions, a program cannot display a proper error message to the user, for example, it might be running as a service that is not allowed to interact with the desktop. Quite often, the user may not be qualified to interpret an error message, and the best way to proceed is to write the error to a known location and have it interpreted by system administrators or other qualified personnel later. Domain administrators can connect to a machine's event log and retrieve entries remotely, without needing to physically walk to the user's machine. That allows for implementing monitoring solutions that can retrieve errors and react to them—and the errors could be generated by programs running anywhere on the network.

The event log infrastructure recognizes the concept of logs, or files that are basically containers for event log entries or messages. A system will have several default logs that were created at installation time, plus zero or more custom event logs. For example, any Windows system has the Application, Security, and System event logs. PowerShell's installation adds another—the Windows PowerShell log.

The primary tool that system administrators use for viewing messages is the Windows Event Viewer control panel. It is available under the Administrative Tools section, which you can reach via both the Start menu and the Control Panel. Figure 16-1 shows a screen shot of the Windows Event Viewer management snap-in.

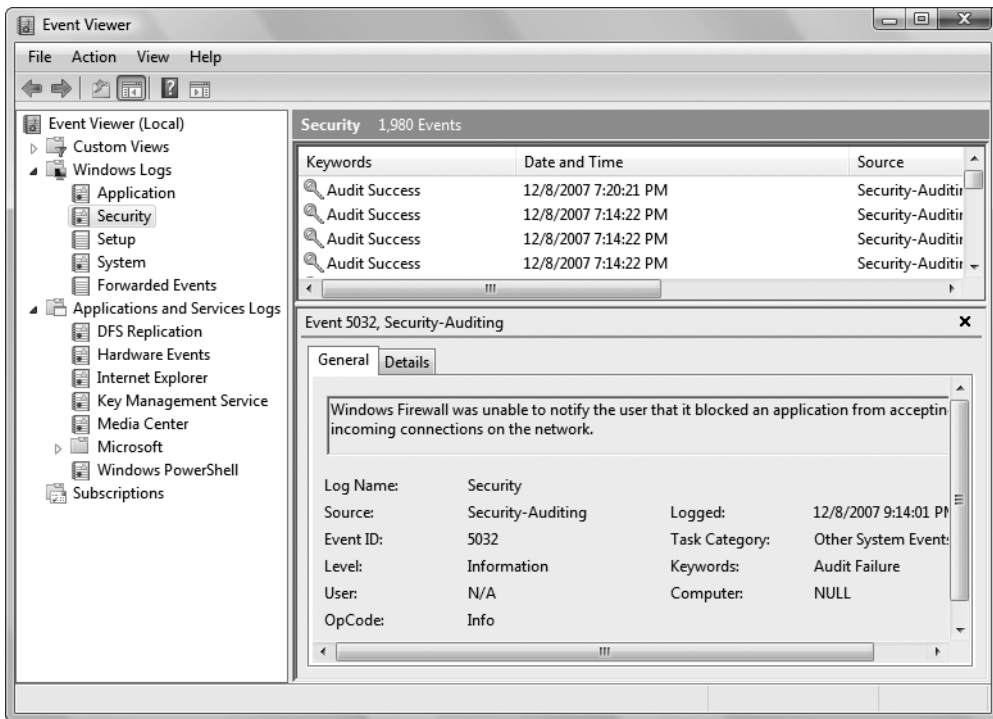


Figure 16-1. The Windows Event Viewer showing event log entries

Reading from the System Event Logs

The Event Viewer is a nice and powerful application, but it does little to help us in our scripting needs. There is no way to use it from a PowerShell script to get event log entries, so fortunately, PowerShell offers a separate cmdlet that is the entry point of all event-log-related operations—Get-EventLog.

The first thing that we need to do is get a list of all event logs on the system. We can do that using the -List cmdlet switch:

```
PS> Get-EventLog -List
```

```
Max(K) Retain OverflowAction      Entries Name
```

-----	-----	-----
20,480	0 OverwriteAsNeeded	630 Application
512	7 OverwriteOlder	0 DFS Replication
20,480	0 OverwriteAsNeeded	0 Hardware Events
512	7 OverwriteOlder	0 Internet Explorer
512	7 OverwriteOlder	0 Key Management Service
8,192	0 OverwriteAsNeeded	0 Media Center
20,480	0 OverwriteAsNeeded	1,980 Security
20,480	0 OverwriteAsNeeded	6,069 System
15,360	0 OverwriteAsNeeded	847 Windows PowerShell

As you can see, Windows Vista has some additional event logs beyond the default ones of System, Security, and Application. The preceding list contains objects that show some of the event logs' attributes, such as the size limits and the number of entries each of them currently contains.

We can read individual entries from an event log by specifying its name as the `-LogName` parameter value. Doing so will return all entries, sorted by the time they occurred in descending order. There is a small catch here, though. If you look at the preceding sample output, you can see that an event log can contain thousands of entries; for example, the System event log contains 6,069 records. Retrieving them can be very slow, and most of the time, we do not need to retrieve all records. The typical usage scenario goes like this, "I need the last 'X' entries, so that I can see if they contain the specific entry that I am looking after."

We can limit the number of entries that we retrieve by using the `-Newest` parameter: it instructs the cmdlet to return the number of events that occurred last. Here is how to use it to get the last ten events in the System log:

```
PS> Get-EventLog -LogName System -Newest 10
```

Index	Time	Type	Source	EventID	Message
-----	-----	----	-----	-----	-----
6070	Dec 08 23:06	Info	Service Control M...	7036	The descript...
6069	Dec 08 22:55	Info	Dhcp	1103	Your compute...
6068	Dec 08 22:50	Info	Service Control M...	7036	The descript...
6067	Dec 08 21:55	Info	Dhcp	1103	Your compute...
6066	Dec 08 20:55	Info	Dhcp	1103	Your compute...
6065	Dec 08 19:55	Info	Dhcp	1103	Your compute...
6064	Dec 08 19:20	Info	Service Control M...	7036	The descript...
6063	Dec 08 19:20	Info	Service Control M...	7036	The descript...
6062	Dec 08 19:14	Info	Service Control M...	7036	The descript...
6061	Dec 08 19:14	Info	Service Control M...	7036	The descript...

Event entries are usually distinguished by their event ID, type, and source. The event ID is the most important property, as it uniquely qualifies an event entry. The entry type is interpreted as the severity of the event that occurred. It can have the value of Information, Warning, or Error. The source contains a string displaying the name of the program or system component that logged the event. We can use that to filter event log entries by piping the results we got from `Get-EventLog` through the `Where-Object` cmdlet. Let's do that and get the Information entries generated from the Dhcp source:

```
PS> Get-EventLog -LogName System -Newest 10 | `
where { $_.EntryType -eq "Information" -and $_.Source -eq "Dhcp" }
```

Index	Time	Type	Source	EventID	Message
----	----	----	-----	-----	-----
6069	Dec 08 22:55	Info	Dhcp	1103	Your compute...
6067	Dec 08 21:55	Info	Dhcp	1103	Your compute...
6066	Dec 08 20:55	Info	Dhcp	1103	Your compute...
6065	Dec 08 19:55	Info	Dhcp	1103	Your compute...

Event logs have their own security settings, and not all event logs are created equal. Let's see what happens when we try to access the Security log:

```
PS> Get-EventLog -LogName Security
Get-EventLog : Requested registry access is not allowed.
At line:1 char:13
+ Get-EventLog <<<< -LogName Security
```

The error is not the most descriptive one on earth, but you may be guessing what is going on already. Only administrators are allowed to retrieve events from the Security log. On Windows Vista, that means that you should run PowerShell with elevated privileges. The Security log contains some interesting information, for example, security audit data (if security auditing has been enabled on the system). And we can use it to get the times of successful user logins on the system (remember to run the following command with administrator privileges):

```
PS> Get-EventLog -LogName Security -Newest 100 | `
where { $_.EventID -eq 4648 -and `
    $_.TimeGenerated.Date -eq [DateTime]::Today } | `
select TimeGenerated,EntryType
```

TimeGenerated	EntryType
-----	-----
12/8/2007 7:20:21 PM	SuccessAudit
12/8/2007 7:14:22 PM	SuccessAudit
12/8/2007 7:12:00 PM	SuccessAudit
12/8/2007 5:48:45 PM	SuccessAudit
12/8/2007 5:47:53 PM	SuccessAudit
12/8/2007 5:44:51 PM	SuccessAudit
12/8/2007 5:38:53 PM	SuccessAudit
12/8/2007 5:36:33 PM	SuccessAudit
12/8/2007 3:27:54 PM	SuccessAudit
12/8/2007 3:19:44 PM	SuccessAudit
12/8/2007 3:11:29 PM	SuccessAudit
12/8/2007 3:03:06 PM	SuccessAudit
...	

The preceding command relies on the fact that user login events have the ID of 4648. To get the correct ID of an event that you want to monitor from within PowerShell, open the Event Viewer application, and get the Event ID value from there. The easiest way to get the ID is to

double-click an event log entry and get the value from the Event Properties window, as shown in Figure 16-2.



Figure 16-2. Event Viewer showing event log entry details revealing the Event ID

It is possible to create scripts that listen to events raised by the system event log and react whenever a specific event log entry is written there. Unfortunately, this is not directly supported by PowerShell, and we need an additional snap-in—PSEventing—to do that. Chapter 22 describes in detail how to subscribe to login events raised by the event log.

Accessing Remote Event Logs

The `Get-EventLog` cmdlet seems to be lacking an important parameter, `-Computer`. It can only access event logs on the local machine. We are out of luck if we want to get an entry from a remote system. There is a way to do that, but it is not as easy as calling a single cmdlet. The easiest approach would be to use the WMI infrastructure. The `Get-WmiObject` cmdlet allows us to retrieve objects from remote machines and get entries from a remote event log. The WMI object that exposes an event log is the `Win32_NTEventLogFile` object. It represents objects similar to the ones we got when we requested the list of event logs from `Get-EventLog`. Actual event log entries are represented by `Win32_NTLogEvent` objects. They have a `LogFile` property that allows us to distinguish between, say, entries stored in the Application log and entries stored in the Security one.

Getting events using WMI presents us with the same problem we had when working with `Get-EventLog`—we need a way to restrict the number of events that we get back from our query, so that we get better performance out of our script. WMI allows us to filter objects that we get back with a special language called WMI Query Language (WQL). We can craft a filter that uses

the event entry `RecordNumber` property to get the last “X” events. To make that work, we need the total number of events. We need to get the `Win32_NTEventLogFile` object for our log file and get the number of entries it contains. We then need to calculate a threshold value by subtracting the number of entries we need from the total number of entries. After that, we can craft a query that returns all `Win32_NTLogEvent` objects with a `RecordNumber` greater than our threshold value. We can demonstrate that from a script, called `Get-RemoteEvents.ps1`, which connects to the NULL computer and gets the last ten entries in the Application event log. Here is the code:

```
$applicationLog = Get-WmiObject Win32_NTEventLogFile `
    -filter "LogFileName = 'Application'" `
    -computer NULL

$startRecord = $applicationLog.NumberOfRecords - 10
$filter = "Logfile = 'Application' AND RecordNumber > $startRecord"
Get-WmiObject Win32_NTLogEvent -filter $filter `
    -computer NULL | `
    sort TimeGenerated -Descending
```

The script returns a collection of objects sorted by their `TimeGenerated` property values in descending order. We can call it on the command line and display the `EventCode`, `SourceName`, and `Message` properties, formatted as a list:

```
PS> .\Get-RemoteEvents.ps1 | select EventCode,SourceName,Message | fl

EventCode : 2004
SourceName : usbperf
Message    : Usbperf data collection failed. Collect function called w
            ith unsupported Query Type.

EventCode : 1008
SourceName : Microsoft-Windows-Perflib
Message    : The Open Procedure for service "WmiApRpl" in DLL "C:\Wind
            ows\system32\wbem\wmiaprpl.dll" failed. Performance data
            for this service will not be available. The first four by
            tes (DWORD) of the Data section contains the error code.

...
```

WMI is a powerful tool that can help with automating all sorts of management activities. Chapter 20 covers in detail what WMI is and how we can use it from within PowerShell.

Writing to the Event Log

Automating event log reads and inspecting data are important management activities, but they are only part of the story. We often need to write to an event log too. The best example for needing to do so is a script that runs as a scheduled task under a different user account and cannot interact with the desktop. That script needs a way to notify users and administrators whenever it performs an operation or faces a problem that it cannot work around. Since it cannot print a message to the screen, it has to write it somewhere. The simplest solution is to use a text file in

a well-known location such as `C:\Windows\Temp\importantscript.log` and document the location somewhere. A better solution is to write an entry to the Application event log. That log was created specifically for user applications and any program can write entries to it.

Writing event log entries is not directly supported by PowerShell, and we need to use a .NET class to help us with that task. The `System.Diagnostics.EventLog` class has been a part of the .NET framework since its initial release. It can read and write entries to the event log, and in fact, the `Get-EventLog` cmdlet uses it to retrieve entries we need.

The first thing that we need when logging an event is an event source, a short string that identifies the application that generated the event. We need to register the event source with the event log system before we use it when logging events. To register our source, we need to call the `[System.Diagnostics.EventLog]::CreateEventSource` static method. There's one minor caveat here—we need administrator privileges to do that. Using the `EventLog` class, we can create a short script called `New-EventSource.ps1` that takes an event source string as a parameter, verifies whether it already exists, and if it doesn't, creates it. Here is the code:

```
param ([string]$source = $(read-host "Enter Event Source"))

function New-EventSource([string] $source)
{
    if(![System.Diagnostics.EventLog]::SourceExists($source))
    {
        [System.Diagnostics.EventLog]::CreateEventSource($source, `
            'Application')
    }
    else
    {
        Write-Warning "Source $source already exists."
    }
}

trap
{
    Write-Error @"
Could not create event source.
Make sure you run this script with Administrator privileges.
"@

    exit
}

New-EventSource $source
Write-Host "Event source $source created in the Application event log"
```

The script assumes that any error raised at the time of creating the event log has been caused by not having enough permissions and writes a friendly error message suggesting that. In addition, the code issues a warning if the event source already exists. To get further information on working with the `Write-Warning` and `Write-Error` cmdlets, please refer to Chapter 9.

Let's now use the script to generate a new event source for our scripts and call it `PowerShellScripts`:

```
PS> .\New-EventSource.ps1 PowerShellScripts
Event source PowerShellScripts created in the Application event log
PS>
```

Now that we have our event source registered, we can log entries without needing to run PowerShell as an administrator. We can create a script, `Write-Event.ps1`, that takes as parameters the message, the event source, the entry type, and the event ID for a log entry and writes that to the Application log. The script has to create an `EventLog` object, configure its `Log` and `Source` properties, and write the entry using the `WriteEntry()` method. We want to make the script both easy to use on the command line and easy to call from another script. That is why the script will prompt the user to enter any of the parameters he or she has forgotten to supply. Here is the code:

```
param
(
    [string]$message = `
        $(read-host "Enter a Event Description"),
    [string]$source = `
        $(read-host "Enter Event Source"),
    [string]$type = `
        $(read-host "Enter Event Type [Information, Warning, Error]"),
    [int]$eventId = `
        $(read-host "Enter EventID")
)

function Write-Event( `
    [string] $message,
    [string] $source = $(throw "Event source parameter required."),
    [System.Diagnostics.EventLogEntryType] $type,
    [int] $eventId)
{
    # Check if the event source actually exists.
    if(![System.Diagnostics.EventLog]::SourceExists($source))
    {
        throw "Event source $source does not exist."
    }

    $log = New-Object System.Diagnostics.EventLog
    $log.Log = "Application"
    $log.Source = $source
    $log.WriteEntry($message, $type, $eventId)
}
```

```
Write-Event $message $source $type $eventId
```

```
Write-Host @"
Logged event:"
```

```
$message
```

```
Source: $source
```

```
Type: $type
```

```
Event ID: $eventId
```

```
"@"
```

As you can see, all the real work happens in the `Write-Event` function. It can easily be taken out and added to a script library and thus reused in other scripts. Here is how to use the script to log a new entry:

```
PS> .\Write-Event.ps1 -message "Test message from PowerShell" `
-source PowerShellScripts -type Information -eventId 1000
Logged event:
Test message from PowerShell
```

```
Source: PowerShellScripts
```

```
Type: Information
```

```
Event ID: 1000
```

Figure 16-3 shows how our entry looks when viewing its properties in the Windows Event Viewer.

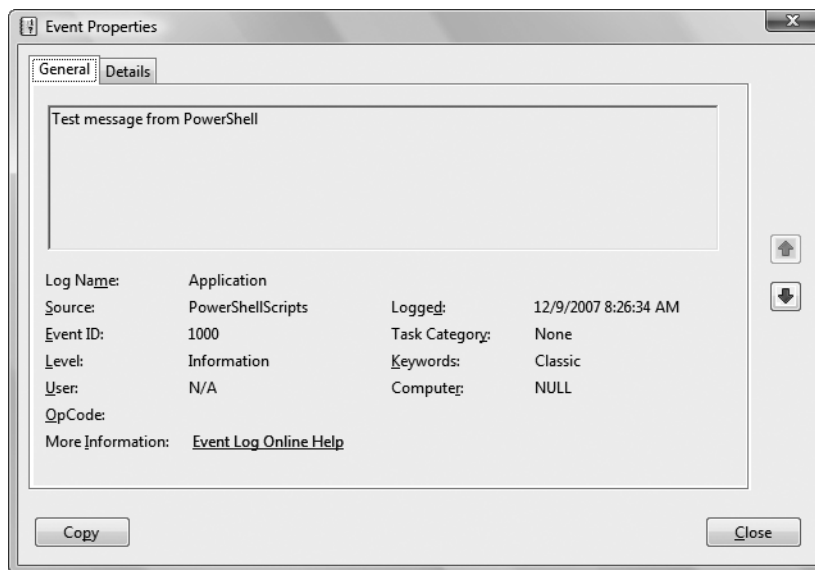


Figure 16-3. The Event Viewer showing our log entry

I picked the event ID value of 1000 at random. When logging real entries, you should carefully pick the event IDs and use them to distinguish among different event types. For example, you can use an Event ID of 1000 for describing one type of error and 1001 for another. Anyone knowing the event source and what different event IDs stand for can easily interpret log entries written by your script.

Performance Counters

Windows performance counters are special objects supported by the operating system that allow applications and components to publish performance data. They also allow other applications to capture and analyze that published data. The performance data can be of any kind: ranging from the standard values of the percentage of CPU time that a program uses or the amount of memory that it has allocated to application-specific metrics like the number of transactions per minute a service is able to process. We can use the data provided by performance counters to identify system bottlenecks and fine-tune our application or system performance.

Performance counter data is available through a control panel application that is called Reliability and Performance Monitor on Windows Vista. The application is a part of the Performance Information and Tools control panel section and can be found inside the Advanced Tools subsection. Previous versions of Windows expose performance counter data through the Performance control panel application located inside the Administrative Tools section. Figure 16-4 shows the Reliability and Performance Monitor application graphing statistical data about the percentage of total CPU time used on the system.

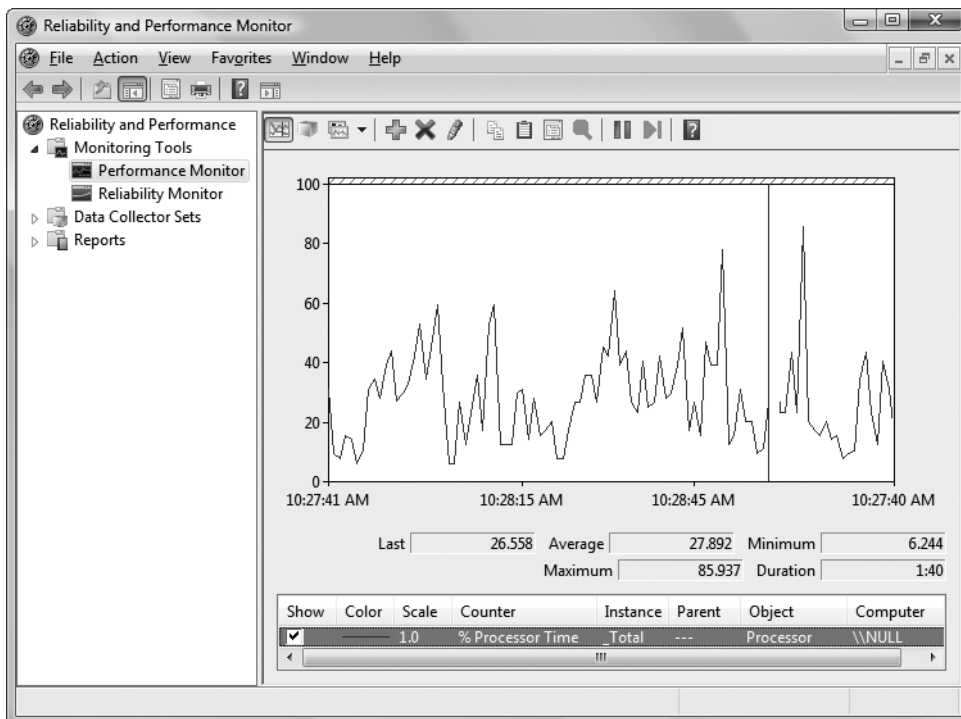


Figure 16-4. The Reliability and Performance Monitor graphing data coming from a performance counter

We can add other counters to be monitored by right-clicking the list of counters in the lower window area and selecting Add Counters from the context menu. We are presented with a dialog box that looks like the one shown in Figure 16-5.

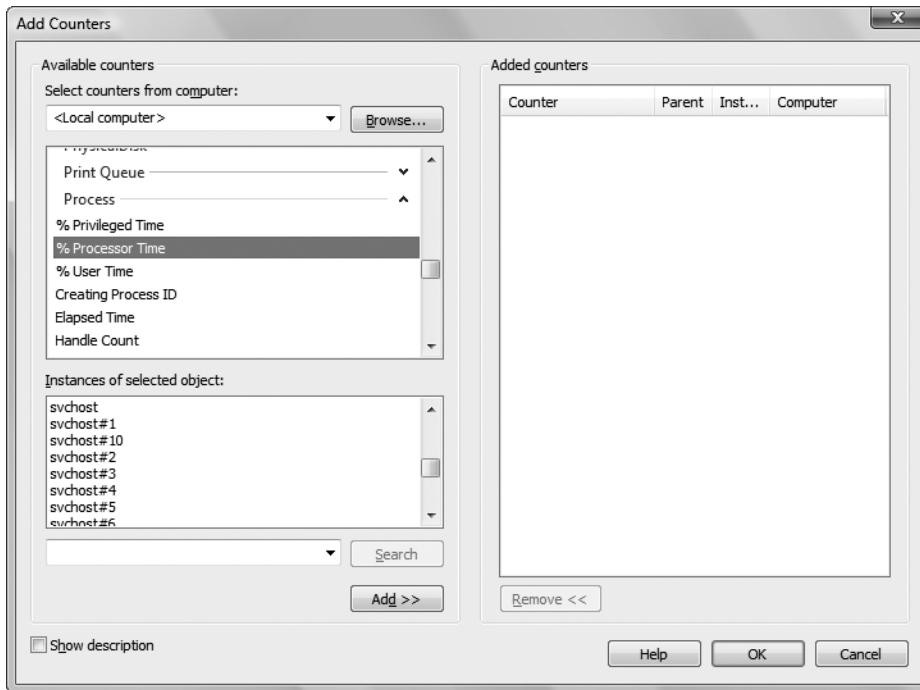


Figure 16-5. Adding a new process performance counter

The dialog box presented in Figure 16-5 is important two reasons. The first is that we can get the performance counter names from the list and use them from our PowerShell scripts. Yes, the names are the same: % Processor Time, Handle Count, and so on. The second important thing is the Instances of selected object list box. When we are gathering process-related performance counter data, counters are named with the name of the process without the .exe suffix. If we have more than one process with the same name, the subsequent counters are named with #1, #2, #3, and so on suffixes. Most of the time, including in the examples presented in this chapter, we will be working with processes that have only a single instance, but if you are faced with a multiple instance situation, you now know that you can get the correct counter for your process in the Add Counters dialog box.

Consuming Counter Data

Unfortunately, PowerShell has no built-in cmdlet that allows us to get performance counter values. We have to use .NET classes for that. Fortunately, it is not too hard—all we have to do is create a `System.Diagnostics.PerformanceCounter` instance, configure its properties, and repeatedly call its `NextValue()` method. Before we start doing that, we should first configure our permissions. Only users that have been added to the Performance Monitor Users group can access performance counter data. Figure 16-6 shows my current user added to that group. Note the text at the bottom-right corner of the window; this detail has bitten me in the past! You need to log off and then log on again for changes to Windows group membership to take effect.

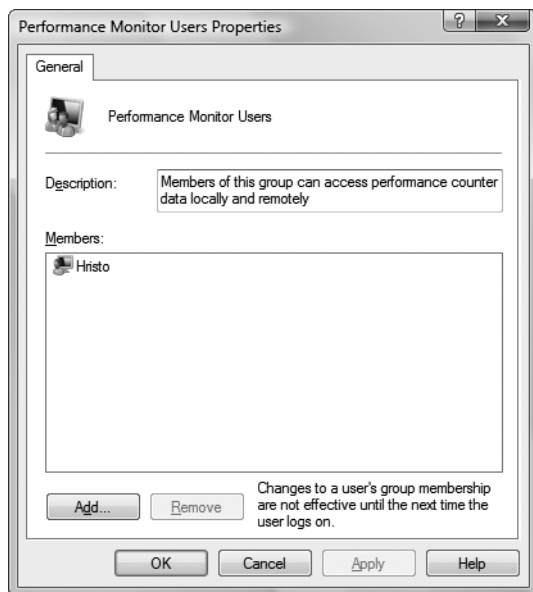


Figure 16-6. *The Performance Monitor Users group members*

Now that our permissions are all set up, we can create a small script, `Monitor-TotalCpuUsage.ps1`, which uses a performance counter to fetch and display the current CPU load every second. The script loops forever, so just press Ctrl+C when you want to terminate it. Here is the code:

```
$counter = New-Object Diagnostics.PerformanceCounter
$counter.CategoryName = "Processor"
$counter.CounterName = "% Processor Time"
$counter.InstanceName = "_Total"

while ($true)
{
    $value = $counter.NextValue()
    Write-Host "CPU: $value"

    sleep 1
}
```

The settings for the category, counter name, and instance name can be easily obtained from the new performance counter dialog box shown in Figure 16-5. Here is what running the script produces:

```
PS> .\Monitor-TotalCpuUsage.ps1
CPU: 0
CPU: 11.27252
CPU: 14.66121
CPU: 14.88817
CPU: 10.15882
```


The CPU does not seem too overworked at the moment; only about ten percent of its resources are being utilized.

Monitoring Programs

We live in an imperfect world where we may have to deal with misbehaving programs. Quite often, we have to use a program that is defective in some way, and it may need to be closely monitored and possibly restarted if it gets hung or consumes too many resources. Performance counters are the perfect tool for that monitoring. Using them, we can create small companion scripts that watch over other programs. They can just sit there, analyzing performance data, and take action when needed.

Detecting Memory Leaks

A very common programming error is keeping references to objects in memory without releasing and deallocating them when needed. This error may manifest in programs as having the memory usage grow and grow over time until the program takes up all available memory. Having no available memory may prevent other programs from starting or working properly. Using a Private Bytes performance counter for the Process category, we can keep track of a process' allocated memory and restart it when it reaches a certain threshold. I have a tiny misbehaving program, called `MemoryHog.exe`, which will do nothing but allocate 2MB of memory every second. We can tame that program with a script called `Monitor-MemoryUsage.ps1`, which will get the allocated memory from a performance counter and terminate the process if it goes above 30MB of allocated memory. Here is the code:

```
$counter = New-Object Diagnostics.PerformanceCounter
$counter.CategoryName = "Process"
$counter.CounterName = "Private Bytes"
$counter.InstanceName = "memoryhog"
```

```
function Get-MemoryUsage()
{
    $value = 0
    trap
    {
        continue
    }
    $value = $counter.NextValue()
    return $value
}
```

```
while ($true)
{
    $value = Get-MemoryUsage
    $valueMegabytes = $value / 1024 / 1024
    Write-Host "Memory: $valueMegabytes"
```

```

    if ($valueMegaBytes -gt 30)
    {
        Write-Host "Process exceeded the memory limit. Terminating..."
        Stop-Process -Name memoryhog
        exit
    }

    sleep 1
}

```

Note the Process counter category and the Private Bytes instance name; I got those from the Reliability and Performance Monitor dialog shown in Figure 16-5. The only special thing about the preceding code is the way we get the performance counter value—we have moved it to a separate function and added an error trap, so that we return a value of 0 if the process has not been run yet. We check the memory usage every second, and terminate the process if the \$valueMegaBytes variable contains a number greater than 30. Here is the output we get after we run MemoryHog.exe from within Windows Explorer and then start the script:

```

PS> .\Monitor-MemoryUsage.ps1
Memory: 19.08203125
Memory: 21.08203125
Memory: 24.29296875
Memory: 26.29296875
Memory: 28.29296875
Memory: 29.66796875
Memory: 31.66796875
Process exceeded the memory limit. Terminating...

```

Detecting Hung Programs

Similar to the high memory usage situation, a program may go astray and fall into an infinite loop. Those programs keep trying to do something, fail, and try again. The result is that they hang and occupy all available CPU resources. When that happens, the only possible solution is to terminate the program.

We can use a CPU-related performance counter and write a script, Monitor-CpuUsage.ps1, that gets the percent of CPU time a process uses every second. If it gets ten consecutive measurements that are above 80 percent, it terminates the program. It uses a counter that gets incremented for every reading that is above 80 and gets zeroed out when we detect a reading below that threshold. The monitored program in this example is a small executable called CPUHog.exe, which will do nothing but loop indefinitely until it occupies all CPU resources. It tries to simulate a program hung in an infinite loop. Here is our monitor script's code:

```

$counter = New-Object Diagnostics.PerformanceCounter
$counter.CategoryName = "Process"
$counter.CounterName = "% Processor Time"
$counter.InstanceName = "cpuhog"

```

```

function Get-CpuUsage()
{
    $value = 0
    trap
    {
        continue
    }
    $value = $counter.NextValue()
    return $value
}

$abnormalMeasureCount = 0

while ($true)
{
    $value = Get-CpuUsage
    Write-Host "CPU: $value"

    if ($value -gt 80)
    {
        $abnormalMeasureCount++
    }
    else
    {
        $abnormalMeasureCount = 0;
    }

    if ($abnormalMeasureCount -gt 10)
    {
        Write-Host "Process appears to be hung. Terminating..."
        Stop-Process -Name cpuhog
        exit
    }

    sleep 1
}

```

Again, I got the % Processor Time counter name from that same Reliability and Performance Monitor dialog. As you can see from the preceding code, we terminate the process only if our \$abnormalMeasureCount variable gets incremented to a value above 10. Here is what we get when we run the CPUHog.exe program from within Windows Explorer and then start the monitor script:

```

PS> .\Monitor-CpuUsage.ps1
CPU: 0
CPU: 94.12367
CPU: 90.34071
CPU: 88.80951
CPU: 90.0821

```

```

CPU: 87.44567
CPU: 89.82497
CPU: 90.68784
CPU: 95.20763
CPU: 81.51465
CPU: 91.95991
CPU: 93.40311
Process appears to be hung. Terminating...

```

Note that the 80 percent CPU usage threshold may need some tweaking in your specific scenario. For example, on a dual-core or a multiprocessor system, that value may be below 50 percent, because the executing process will hog only one of the CPU cores, leaving the rest available.

Detecting Unexpected Program Exits

While we are on the topic of monitoring external programs, I'll cover a scenario that involves a broken program that may exit unexpectedly. We do not need a performance counter to resolve that, but we still need a monitor script. We will create a script, `Monitor-Crashes.ps1`, that will start a process and wait for it to exit. We are relying on the fact that PowerShell will block script execution if we execute an external program until that program exits. Our script will try to restart the program five times and then give up. Here is the code:

```

function Start-Process
{
    Write-Host "Starting process..."
    .\UnpredictableCrash.exe
}

for ($i = 0; $i -lt 5; $i++)
{
    Start-Process

    Write-Host "Process exited $($i + 1) times."
}

Write-Host "Program restart limit exceeded."

```

Here is the output we get after running the script:

```

PS> .\Monitor-Crashes.ps1
Starting process...
Process exited 1 times.
Starting process...
Process exited 2 times.
Starting process...
Process exited 3 times.
Starting process...

```

```
Process exited 4 times.  
Starting process...  
Process exited 5 times.  
Program restart limit exceeded.
```

We are using a small program, `UnpredictableCrash.exe`, which pretends it is doing something for a second and then exits. This way, we simulate a program that may be doing useful work but crashing under some obscure circumstances. You can easily change the script so that it restarts the program if the program must be running at all times.

Summary

Windows provides nice utilities that allow us to consume event log entries and performance counter data in a friendly way using a convenient graphical user interface. Sometimes, though, we need to get hold of that data from within a script that tries to accomplish a task depending on an external program. Fortunately, PowerShell and .NET allow us to get that data relatively easily and then move on to concentrate on the task we should really be doing.

In this chapter, I demonstrated how to get entries from the system event logs and how to write your own entries inside. I then outlined several techniques for monitoring applications using Windows performance counters. Using both the event logs and performance counters, we can turn PowerShell into a formidable system-monitoring tool that can save us a lot of work and time spent manually inspecting the graphical tools that Windows already offers.



PowerShell and the World Wide Web

“The Internet” is a loose term and can mean many things to different people. Some of those meanings, in no particular order, can be the World Wide Web, e-mail, file transfers, and P2P transfers like Bit Torrent, instant messaging, and so on. Most of what we know as the Internet today is built on top of the Hypertext Transfer Protocol, or HTTP for short, which is a request/response protocol that exchanges data between a client and a server. While originally designed to publish and retrieve web pages, it is now used to do a lot more, including transporting all sorts of media, news feeds, and even serving as a medium to exchange messages between programs running on different servers.

You might be wondering what a console shell may have in common with the Internet, and the World Wide Web in particular? Every once in a while, we may have to deal with a remote resource—maybe to download or upload a file or just check if a server is up and running. This is seldom needed from an interactive shell session, but it can be tremendously useful in scripts that automate repetitive tasks. PowerShell does not have any built-in features that can help us with downloading or uploading files or even working with web pages. Fortunately, the .NET Framework provides excellent support for working with the HTTP protocol. It contains classes that allow us to work with files located on remote servers that provide an easy-to-use interface while hiding the gory details of working with raw connections.

In this chapter, we will start with the basic task of downloading files via the HTTP protocol. Once you know how to do that, we will process downloaded web pages from our scripts and extract useful pieces of information from them. I will show you how easy it is to work with news feeds exposed from remote sites by downloading the feed XML data and processing it from within our scripts. Last but not least, we will be calling web services to communicate with programs on remote servers and send commands or fetch data. As an added bonus, we will show how to work with File Transfer Protocol (FTP) connections and upload and download files with that protocol.

Laying the Foundation

Traditionally, working with the web and downloading and uploading files to remote computers has been out of the scope of a shell’s functionality. It is good programming practice to lay a clear boundary between software that starts and manages other software (the shell) and software that works with files on remote servers. There are popular, free programs with formidable

feature lists that can do that very well; the first that come to mind are `wget` and `cURL`. Both of these are console applications that have sprung out of the UNIX world, but their Windows ports are fully functional and can both download and upload files over several protocols. If you have invested in scripts that rely on such utilities, you can easily port them to PowerShell; it fully supports working with console applications.

Working with external programs has one drawback—those programs have to be installed on the machine where our scripts will be running. Installing software is easy on only one workstation, but it can quickly become a major hassle if we have to distribute the script to several computers. Fortunately, there is a better way: we can use the .NET Framework to do our job without resorting to external programs. The framework ships with the `System.Net.WebClient` class that is both very powerful and easy to use. It can handle all the uninteresting details of establishing and maintaining an HTTP or FTP session and allow us to focus on the actual results.

Fetching Files from the Web

The original intent behind the HTTP protocol was to allow people to easily share hypertext documents. Documents are identified with a Uniform Resource Locator (URL) and can contain links to other documents, multimedia, and whatnot. A web browser works by downloading a document and analyzing the links to other resources, and in turn, downloading those to the client computer. After that, it assembles and displays the web page to the user. We can do the same in PowerShell and skip the actual display.

Getting a web page's contents is very easy once we have the `WebClient` class at our disposal. Here is how we can use it to get the contents of the Yahoo! start page:

```
PS> $client = New-Object System.Net.WebClient
PS> $client.DownloadString("http://www.yahoo.com")
<html>
<head>
<title>Yahoo!</title>
...
```

Yes, two lines are all we need! The first one creates an instance of the `WebClient` class, and the second one uses the `DownloadString` method on that newly created object to get the contents of the web page as a string. The string that we get back is the HTML code of the web page. HTML (Hypertext Markup Language) is a somewhat human-readable text format used to describe web pages' contents to browsers for display to the user. We will work with that code and extract information from it later on in this chapter.

We can now use what you learned in the preceding example and wrap that code in our first utility script called `Get-Url.ps1`. It will take a single string as a parameter and return the contents as a string. Here is the code:

```
param ($Url)
$client = New-Object System.Net.WebClient

$content = $client.DownloadString($Url)
$content
```

We make sure we output the `$content` variable so that it is available as a result or as a value further down the pipeline. Here is how we can use that script:


```
PS> .\Get-Url.ps1 "http://www.yahoo.com"
<html>
<head>
<title>Yahoo!</title>
...
```

Let's now try something fancy and download a binary file. There are tons of pictures on the Web, but let us try our luck with Flickr's logo, located at http://l.yimg.com/www.flickr.com/images/logo_home.png:

```
PS> $url = "http://l.yimg.com/www.flickr.com/images/logo_home.png"
PS> .\Get-Url.ps1 $url
%PNG

IHDR          tEXtSoftware Adobe ImageReadyq<  -IDATx
...
```

This feels like success, but somehow, it is not very useful. That binary representation does not look too good on the console. How do we properly view that image? Of course, we can save it to a file by piping the result to the Set-Content cmdlet:

```
PS> .\Get-Url.ps1 $url | Set-Content flickr-logo.png
PS> Invoke-Item flickr-logo.png
```

We use the Invoke-Item cmdlet to make Windows open the newly saved file. Figure 17-1 shows how it looks in the default Windows Photo Gallery application that comes with Windows Vista.



Figure 17-1. Viewing our newly downloaded image

Now, piping the downloaded string through Set-Content may not be the best way to handle saving a downloaded file, and it is certainly not the shortest. We can do better here by using the WebClient object's DownloadFile method to save the file directly to disk. We can incorporate that in our script by supporting another parameter: FileName. If it has been provided, the script will pass it to DownloadFile, and that method will save the file contents. Here is the modified code:

```
param ($Url, $FileName)

$client = New-Object System.Net.WebClient

if ($FileName)
{
    $client.DownloadFile($Url, $FileName)
}
else
{
    $contents = $client.DownloadString($Url)
    $contents
}
```

Our job is quite simple here; the DownloadFile method does everything for us. Here is how we can now call our script:

```
PS> $file = Join-Path (Get-Location) "flickr-logo2.png"
PS> .\Get-Url.ps1 -Url $url -FileName $file
PS> diff (Get-Content flickr-logo.png) (Get-Content flickr-logo2.png)
PS>
```

Note that we need an absolute path that contains both the path and the file name for the resulting file. We can obtain a full path that points to the current directory by using Join-Path and pass the current folder and the file name. I have used the diff command in the last line just to make sure that the file I got from the DownloadFile method is the same as the one I got from DownloadString. There are no differences, and our download logic seems to be working fine. Why do we need to modify our script and make it more complex if we get the same result? The answer is simple—performance. The piped version of our file download script needs to get the entire file contents and then save the file to disk. Imagine doing that for a 4GB DVD image file; you are most likely to get an out-of-memory error. The DownloadFile method is smart enough to write data piece by piece as it arrives from the server, without unnecessarily holding it in memory.

Setting Connection Options and Debugging Connection Problems

The HTTP protocol is designed to handle a lot of situations, and as such, it has become quite complex. It supports all sorts of content and runs over heterogeneous networks. Often, we

need to do extra work to get a network resource, for example, pass extra information in the request headers, go through a proxy server, or authenticate with a set of credentials. In this section, you will learn how to do that, and I will show you the tools that can help craft the right request in unfamiliar situations.

Working with Proxy Servers

It is not uncommon for a computer not to be directly connected to the Internet. Either because of technical restrictions or as a matter of organizational policy, very often connections to web pages outside the local network are established through proxy servers. This is done for all sorts of reasons: the proxy server can cache content to reduce page load time; it can track user activity and even deny access to some resources. Apart from learning how to configure a script so that it uses a proxy server, you will learn how to use one to trace what is sent over the network for debugging purposes.

FIDDLER: A DEBUGGING PROXY SERVER

A debugging proxy server is a program that acts like a real proxy server. It sits between other programs and the real server and forwards data back and forth. Its real job, though, is to log what gets sent over the wire and allow us to inspect the HTTP session. Fiddler is one such proxy server. It is probably the best tool to analyze HTTP traffic and is freely available. The latest release of the tool is version 2, and it is available from <http://www.fiddler2.com>. We will use Fiddler throughout this chapter to inspect what gets sent over the network and to diagnose connection problems. The tool is a real gem and a huge time saver.

After installing Fiddler, you will get a fully functional HTTP proxy server that listens on port 8888 by default. For security reasons, it accepts connections originating from the local machine only; that will be more than enough for our experiments. Fiddler should already have taken care of configuring Internet Explorer to forward all connections to the proxy. For other browsers, like Mozilla Firefox and our own PowerShell scripts, we will have to do that manually; we need to specify `localhost` as the proxy server and 8888 as the port.

Figure 17-2 shows a sample Fiddler session. The user interface is split into two parts. The left pane shows the HTTP connections that the tool has detected and allows you to select a connection for detailed inspection. The right pane shows details of the selected connection. We will be working with only the session inspector, as we will not be interested in fiddling with data or analyzing performance statistics. The session inspector's top pane shows the request details in various formats and detail levels. We can select to view the headers sent to the server, the HTTP headers, any POST data, authentication details, and possibly submitted XML data. The session bottom pane shows details about the response we got from the server. Again, it can display response headers, but it also allows us to view the response body interpreting it as various different formats: text, raw binary, image, and XML.

Figure 17-2 shows a request to Google's search page that was opened from within Internet Explorer. We are viewing both the request and response in the raw text view. Google detected that we are using a modern browser and served the response in a gzip-compressed format. Fiddler fully supports that and allows us to view the raw uncompressed text response.

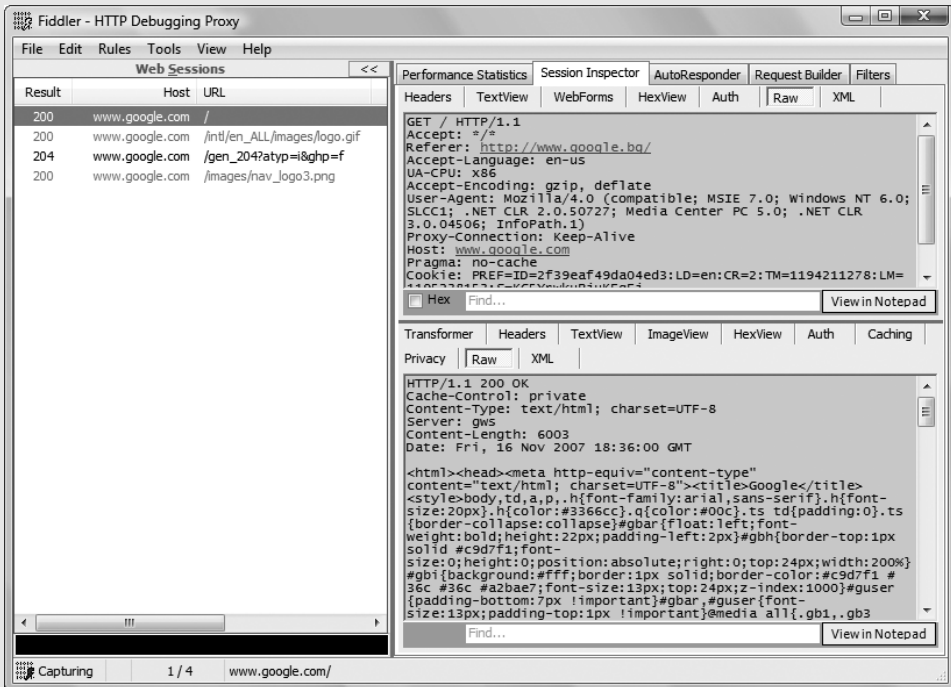


Figure 17-2. The Fiddler session inspector showing a request to Google's search page

How do we configure a proxy for our PowerShell scripts? You might have guessed it—System.Net.WebClient has a Proxy property that we need to initialize with the right object. The object that we must associate with that property is a System.Net.WebProxy one. We need to initialize it with a URL containing the address and port for our proxy server. When using Fiddler, this setting must be `http://localhost:8888`. Let's add proxy server support to our `Get-Url.ps1` script by introducing another parameter, `$Proxy`. We will follow the familiar pattern: if the parameter is not empty, we will set up a new WebProxy object. Here is the new code:

```
param ($Url, $FileName, $Proxy)

$client = New-Object System.Net.WebClient

if ($Proxy)
{
    $proxyConfig = New-Object System.Net.WebProxy -arg $Proxy
    $client.Proxy = $proxyConfig
}
```

```

if ($FileName)
{
    $client.DownloadFile($Url, $FileName)
}
else
{
    $contents = $client.DownloadString($Url)
    $contents
}

```

Two lines of real code and we have proxy server support! The WebClient class is a really nice tool. We can use the script to get the Flickr logo once again:

```

PS> $url = "http://l.yimg.com/www.flickr.com/images/logo_home.png"
PS> $file = Join-Path (Get-Location) "flickr-logo2.png"
PS> $proxy = "http://localhost:8888"
PS> .\Get-Url.ps1 -Url $url -FileName $file -Proxy $proxy
PS>

```

Nothing to see at the console, but this time, we can flip to the Fiddler session inspector for details (see Figure 17-3).

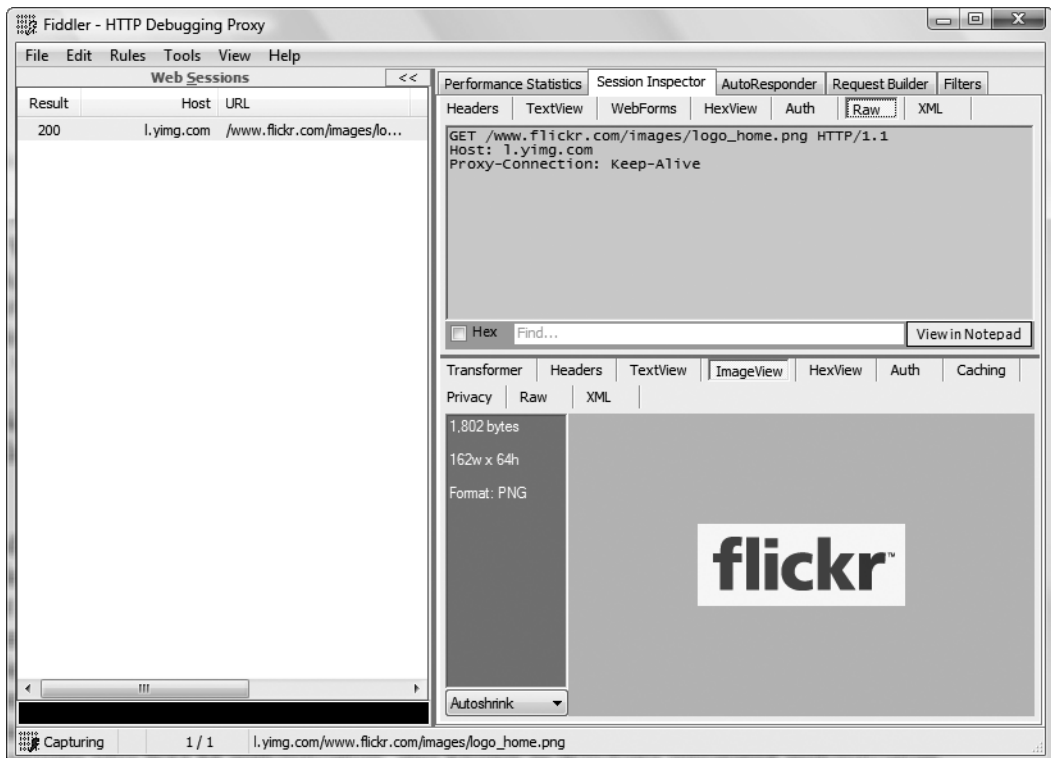


Figure 17-3. Fiddler showing an image returned by the server

Figure 17-3 shows a raw view of our request. As you can see, our PowerShell-initiated web request has fewer headers and a lot less data than the one started by Internet Explorer. The figure also shows one of Fiddler's killer features—the Image view of the response. We can see the actual picture the server returned to us without having to save the data to an external file or switch to another program.

Working with Request Headers

Many important HTTP options and some important data are sent through request headers. Headers are name-value pairs that are transmitted in the request and carry additional information about interpreting the request body. Headers are used to identify the client or the browser to the server, negotiate supported content types, configure caching options, set cookies, and a lot more.

Let's now do a simple request to the Google search page similar to the one we did with Internet Explorer when we introduced Fiddler. Here is the command:

```
PS> $url = "http://www.google.com"
PS> $proxy = "http://localhost:8888"
PS> .\Get-Url.ps1 -Url $url -Proxy $proxy
<html><head><meta http-equiv="content-type" content="text/html; charse
t=ISO-8859-1"><title>Google</title><style>body,td,a,p,.h{font-family:a
...
```

I mentioned that Google knows what type of browser is requesting its pages and serves different content types. This is known as content negotiation. When we visited the page with Internet Explorer, we got back a gzip-compressed version of the page. When we requested the same page from our PowerShell script, we got back a plain-text version. How does Google know what type of content to serve back? The mechanism for doing that relies on HTTP headers. First, Google checks the Accept-Encoding header; it tries to detect if the client supports the gzip or deflate encoding algorithms. Due to buggy browsers that claim that they support gzipped content but fail to actually do so (the first one that comes to mind is Internet Explorer 6.0), Google also checks the User-Agent header. The User-Agent header uniquely identifies the client browser type. At the end, Google streams compressed content to known good browsers that have also indicated that they can handle that type of content.

Let's now try to get the compressed version of Google's search page. First, we will need to add support for sending HTTP headers to our Get-Url.ps1 script. We pass the headers in a Hashtable object via the \$Headers parameter. Our script iterates over the keys of the Hashtable—those will be the header names—gets the associated values, and adds them to the WebClient's Headers collection. Here is the code:

```

param ($Url, $FileName, $Proxy, $Headers = @{})

$client = New-Object System.Net.WebClient

if ($Proxy)
{
    $proxyConfig = New-Object System.Net.WebProxy -arg $Proxy
    $client.Proxy = $proxyConfig
}

foreach ($headerName in $Headers.Keys)
{
    $client.Headers.Add($headerName, $Headers[$headerName])
}

if ($FileName)
{
    $client.DownloadFile($Url, $FileName)
}
else
{
    $contents = $client.DownloadString($Url)
    $contents
}

```

Note how we initialize the `$Headers` parameter to an empty Hashtable. This way, we will not have to check for null values and the `foreach` loop will execute zero times, effectively doing nothing if the parameter has been omitted. Let's now call our script. We need to provide both the `Accept-Encoding` and `User-Agent` headers. The `User-Agent` header must contain the "Mozilla/4.0 (compatible; MSIE 7.0)" string to make Google think it is dealing with an Internet Explorer 7 browser. Here is the actual invocation:

```

PS> $url = "http://www.google.com"
PS> $proxy = "http://localhost:8888"
PS> $headers = @{"Accept-Encoding" = "gzip, deflate"; `
"User-Agent" = "Mozilla/4.0 (compatible; MSIE 7.0)"}
PS> .\Get-Url.ps1 -Url $url -Headers $headers -Proxy $proxy > $null
PS>

```

We redirect the output to `$null` to avoid seeing the gibberish that is actually the binary compressed data. We can view the data in Fiddler. Figure 17-4 shows the initial display of the session inspector. You can see that the request headers contain the `Accept-Encoding` and `User-Agent` values that we provided. For the response part, Fiddler detects that it has received compressed content and offers us the option to decompress it to view the raw text data.

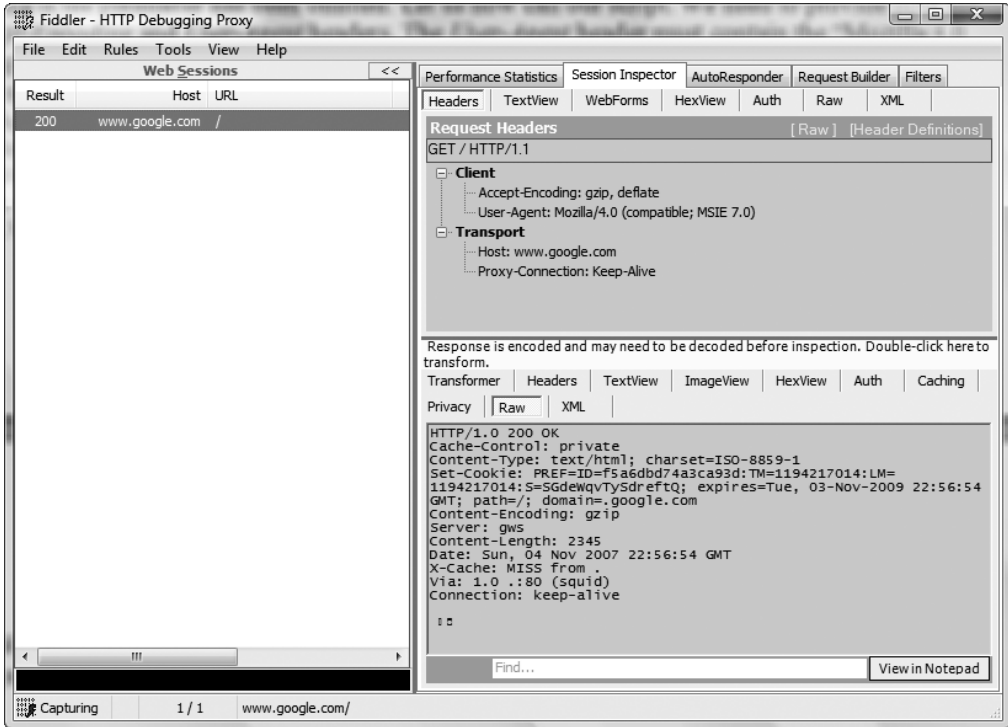


Figure 17-4. Fiddler showing a gzip-compressed response from Google

Accessing Resources Requiring Authentication

Security plays an important part in any network, and quite often, we need to work with files that require authenticating before accessing them. The .NET Framework and System.Net.WebClient, for that matter, support the World Wide Web Consortium–defined HTTP authentication mechanisms plus NTLM (NT LAN Manager) authentication, also known as Windows-integrated authentication. We really need minimal effort to configure our client with the proper credentials.

Before setting up credentials, let's first see what happens if we try to access a protected resource without passing the correct credentials. We will use the test page from <http://www.pagetutor.com> for demonstration purposes; the protected area is at the <http://www.pagetutor.com/keeper/mystash/secretstuff.html> URL. Let's first request it with a normal browser like Internet Explorer. Figure 17-5 shows the password prompt that we get.

Let's see how System.Net.WebClient handles that:

```
PS> $url = "http://www.pagetutor.com/keeper/mystash/secretstuff.html"
PS> .\Get-Url.ps1 $url
Exception calling "DownloadString" with "1" argument(s): "The remote s
erver returned an error: (401) Unauthorized."
At Z:\18 - The Internet\Get-Url.ps1:22 char:39
+     $contents = $client.DownloadString( <<<< $url)
```




Figure 17-5. Internet Explorer shows this password prompt when a page requiring authentication is requested.

The `System.Net.WebClient` class is not a user interface component, and it cannot react with a password prompt. The best it can do is throw an error at us. From the error message, we can see the HTTP status code of 401, which means that we are not authorized to access the resource.

We will now extend our `Get-Url.ps1` script with support for HTTP authentication. To do that, we have to create a `System.Net.NetworkCredential` object, initialize it with the user name and password, and pass that object to the `Credentials` property of the `WebClient` object. We will extend the script with two additional parameters: `$UserName` and `$Password`. The script will create the `NetworkCredential` object if and only if both of those parameters have been provided. Here is the code:

```
param ($Url, $FileName, $Proxy, $Headers = @{},
        $UserName, $Password)

$client = New-Object System.Net.WebClient

if ($Proxy)
{
    $proxyConfig = New-Object System.Net.WebProxy -arg $Proxy
    $client.Proxy = $proxyConfig
}

foreach ($headerName in $Headers.Keys)
{
    $client.Headers.Add($headerName, $Headers[$headerName])
}
```

```

if ($UserName -and $Password)
{
    $credentials = New-Object System.Net.NetworkCredential `
        -argumentList $UserName, $Password
    $client.Credentials = $credentials
}

if ($FileName)
{
    $client.DownloadFile($Url, $FileName)
}
else
{
    $contents = $client.DownloadString($Url)
    $contents
}

```

The `pagetutor.com` page has been set up with a required user name of “jimmy” and a password of “page.” Here is how we can access the password-protected page:

```

PS> $url = "http://www.pagetutor.com/keeper/mystash/secretstuff.html"
PS> .\Get-Url.ps1 -Url $url -UserName "jimmy" -Password "page" `
    -Proxy "http://localhost:8888"

```

```

<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY BGCOLOR="#666666" TEXT="#EEEEEE">

<H2 ALIGN="center">MY SECRET STUFF</H2>

</BODY>
</HTML>

```

As you can see from the response, our request proved successful. Let’s now see what gets transported over the wire. Figure 17-6 shows Fiddler’s view of the request.

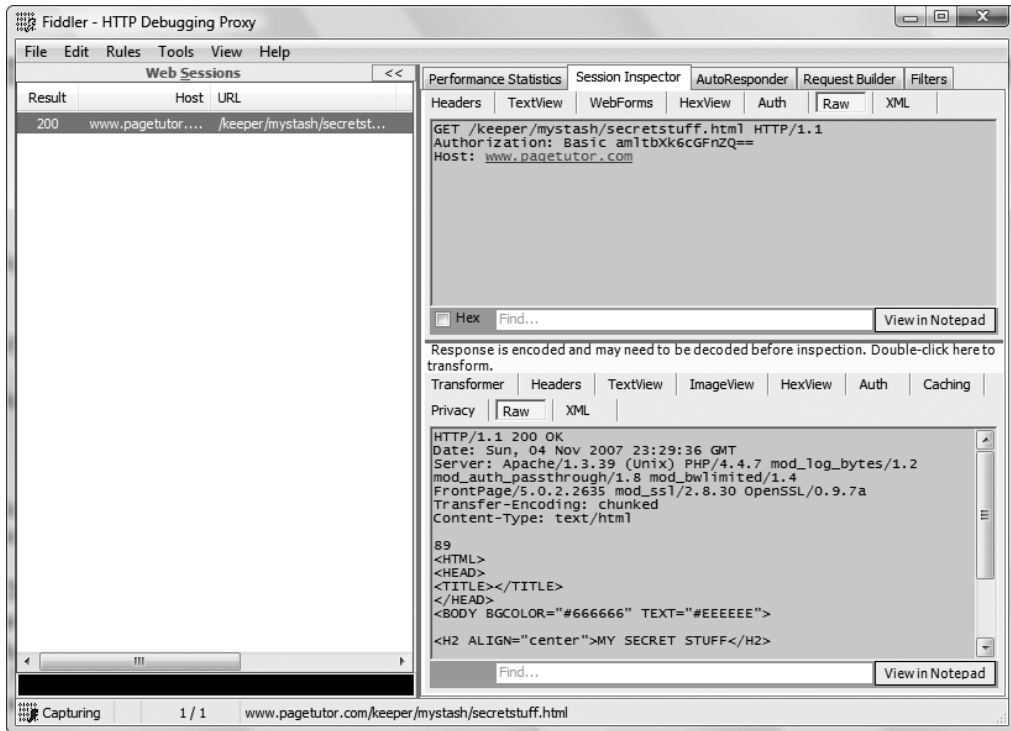


Figure 17-6. Fiddler showing a request to a password-protected page

HTTP BASIC AUTHENTICATION IS NOT SECURE!

The previous example uses the so-called HTTP Basic authentication mechanism. Looking at the Fiddler screen shot, you might be guessing already that the authentication mechanism is implemented with one HTTP header. In our case, this is the `Authorization` header. There is a security risk with that authentication mechanism: the user name and password are sent unencrypted over the network! The credentials are not really sent in plain text; they are base64 encoded. That makes little difference as base64 is not an encryption algorithm and anyone can decode the password back. That makes the algorithm vulnerable to replay attacks. A malicious individual can capture the request data and use the `Authorization` header contents to craft requests that will get authenticated. There is no need for him or her to even decode the actual user name and password—all that is needed is providing the correct `Authorization` header value. Let's play the bad guys and craft a malicious request ourselves:

```
PS> $url = "http://www.pagetutor.com/keeper/mystash/secretstuff.html"
PS> $proxy = "http://localhost:8888"
PS> $headers = @{"Authorization" = "Basic amltbXk6cGFhZQ==" }
PS> .\Get-Url.ps1 -Url $url -Proxy $proxy -Headers $headers
<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
```

```
<BODY BGCOLOR="#666666" TEXT="#EEEEEE">
```

```
<H2 ALIGN="center">MY SECRET STUFF</H2>
```

```
</BODY>
```

```
</HTML>
```

I have copied and pasted the Authorization header contents from Fiddler just as a real attacker would do. The result is disturbing; my request got authenticated without me having to provide the user name or password. If we really wanted to decode the original user name and password, we could certainly do so using the Convert class to get the binary data and passing that to the UTF8 encoding object that can turn it into a string:

```
PS> $encoding = [System.Text.Encoding]::UTF8
PS> $encoded = "amltbXk6cGFnZQ=="
PS> $encoding.GetString([System.Convert]::FromBase64String($encoded))
jimmy:page
```

The bottom line is that the HTTP Basic authentication mechanism is inherently insecure, and you had better resort to the more secure mechanisms like Digest authentication or NTLM.

Testing and Validating Web Sites

Downloading content from a remote server is only the beginning. We have to do something with that content afterward. You can use what you have learned so far to build various utilities that periodically download newer versions of given pages or crawl a web site to find and report broken links. This section will demonstrate extracting pieces of information from web pages' HTML code that we can use later.

Test-Url: Verifying a Page's Existence

Here is an interesting problem: how do we verify that a page or a file exists on a remote server? If we could do that, we could use it to create scripts that run periodically making sure that important files are accessible from the Web. We can also check a dynamically generated list of files, say, all files linked from a document. The only problem is that getting the entire file just to confirm that it exists is wasteful, can be slow, and can put too much stress on the web server. HTTP defines a method, HEAD, designed exactly for that. A HEAD request is similar to the regular GET requests we have seen so far. The only difference is that the server sends back the response headers and does not send the actual file content. This makes it possible for us to write a lightweight script that checks if a file exists on a remote server.

Unfortunately, we cannot use the WebClient class here. The class supports making requests using different HTTP methods, but that feature has been intended as a means to upload data to remote servers. WebClient will always send a request body, and that is incompatible with a HEAD request, which should not have a body according to the HTTP protocol specification. To create a HEAD request, we have to go a bit deeper and use the System.Net.WebRequest class. The

System.Net.WebClient class uses System.Net.WebRequest internally, and we can certainly do that too. We need to create a new WebRequest object by passing the URL in the constructor, set its Method property to "HEAD," and check the response status code. If the code is different than 200 (OK), we will get an error that we need to trap. We will save that piece of functionality in a reusable script that we will call Test-Url.ps1. Here is the code:

```
param ($url)

trap
{
    Write-Host $_
    #clean up the current request
    $request.Abort()
    #an error occurred. ignore it and continue to returning false.
    continue
}

$request = [System.Net.WebRequest]::Create($url)
$request.Method = "HEAD"
$request.Timeout = 5000
if ($request.GetResponse().StatusCode -eq "OK")
{
    #clean up the current request
    $request.GetResponse().Close()
    return $true
}

#should get here only if we got an error for our web request
return $false
```

The line that will raise an error if the server or file does not exist is the one containing the GetResponse method call. If that happens, the error will get suppressed by our trap, and execution will continue after the if statement. That will make the script return \$false. Otherwise, we will execute the return statement inside the if body, and that will exit the script returning \$true. It is very important that we close the response when we are done and, just in case, abort the request if an error occurs. If we do not do that, we will end up with a bunch of hung requests that have not been cleaned up. It seems that the WebRequest class uses an internal connection pool and failing to properly clean up will cause subsequent requests to fail. Finally, here is how we can use the script:

```
PS> .\Test-Url.ps1 http://www.google.com
True
PS> .\Test-Url.ps1 http://www.google.com/nosuchfile
False
PS> .\Test-Url.ps1 http://www.googlec.om/
False
```

Checking If a Page Contains Broken Links

It is time for us to do something bigger and come up with something useful. A common problem with all web sites is that page content gets old over time. Files get moved; pages get relocated. The final result is that links get broken. Manually going over a large site that contains thousands of links is out of the question. We have to automate this task, so that we can schedule it to run periodically and proactively inform us about broken links. It is always best if you get a notification about a problem before a customer browsing your site reports it.

So, here is the plan: We will get an HTML page from the remote server and try to extract all links. We will go over the HTML code of the page and get the href attribute values for all <a> tags. After we do that, we will be able to feed the collected URLs to the Test-Url.ps1 script and generate a link validity report. Let's get started!

We will call the script that gets the list of links from a remote page Get-Links.ps1. It is a short script that features a really obscure regular expression. Here is the code:

```
param ($Url)

$client = New-Object System.Net.WebClient
$contents = $client.DownloadString($Url)

$pattern = [regex] "<a href=(?<quote>[`'"])(?<url>[!`'\">"]+)"
$matches = $pattern.Matches($contents)

$matches | foreach { $_.Groups["url"].Value }
```

The part that gets the contents of the page identified by the \$Url parameter should be familiar. The script extracts the links by matching all occurrences of the <a href="..." or <a href='...' strings. The (?<quote>) and (?<url>) portions of the regular expressions are named groups. We keep track of the opening quote symbol, because we need to refer to it later when we match the attribute value close quote. We do that to avoid the situation of incorrectly closing a single quote with a double quote or vice versa. We collect all symbols inside the quoted attribute value in the url named group. We then use the Match object's Groups property to extract the attribute value. Here is what we get when we run this script on the PowerShell blog's start page:

```
PS> .\Get-Links.ps1 http://blogs.msdn.com/PowerShell/
/powershell/archive/tags/ApartmentState/default.aspx
...
http://blogs.msdn.com/PowerShell
http://www.microsoft.com/technet/scriptcenter/hubs/msh.mspx
...
```

We get both relative and absolute URLs. Before verifying if they point to something real, we need to normalize them by turning the relative URLs to absolute. There are two types of relative URLs that we need to handle: those relative to the current server (the ones starting with a forward slash, /) and those relative to the current page (the ones that do not start with a forward slash or http://). To make the former absolute, we need to prepend them with the server URL. To fix the latter, we need to prepend them with the current page URL. To do that, we introduce two parameters—\$BasePage and \$BaseServer—and prepend one or another depending on the original link value. This is what the code looks like now:

```

param ($Url, $BasePage, $BaseServer)

$client = New-Object System.Net.WebClient
$contents = $client.DownloadString($Url)

$pattern = [regex] "<a href=(?<quote>[`"'])(?<url>[^^"']+)\k<quote>"
$matches = $pattern.Matches($contents)

$links = ($matches | foreach { $_.Groups["url"].Value })
$links | foreach {
    if ($_ -like "http://*")
    {
        return $_
    }
    elseif ($_ -like "/*")
    {
        return "$BaseServer$_"
    }
    else
    {
        return "$BasePage$_"
    }
}

```

Running it for the PowerShell blog, we get this:

```

PS> $url = "http://blogs.msdn.com/PowerShell/"
PS> .\Get-Links.ps1 -Url $url -BasePage $url `
    -BaseServer http://blogs.msdn.com

http://blogs.msdn.com/powershell/archive/tags/ApartmentState/default.
aspx
...

```

This time, we get all links as absolute URLs. We used the `$url` variable as both the document URL and as the base page. Those values would have been different if we had a page name after the slash. For example, the base page for `http://www.example.com/example.html` would be `http://www.example.com/`.

It is about time we feed the links to `Test-Url.ps1`. We do that with the `foreach` cmdlet:

```

PS> .\Get-Links.ps1 -Url $url -BasePage $url `
    -BaseServer http://blogs.msdn.com | `
    foreach { $_, (.\Test-Url.ps1 $_) }

http://blogs.msdn.com/powershell/archive/tags/ApartmentState/default.a
spx
True
http://blogs.msdn.com/powershell/archive/tags/CMDLET/default.aspx
True
...

```

```

The remote server returned an error: (405) Method Not Allowed.
http://imdb.com/title/tt0101669/quotes
False
...
http://blogs.msdn.com/powershell/archive/tags/Impersonation/default.as
px
True
PS>

```

The foreach block outputs an array of two elements for every URL being checked: the URL and the check result. The only URL that fails our check in the preceding code is the one pointing to the `imdb.com` site. Looking at the error message, the resource pointed by the URL is there; it just happens that the `imdb.com` HTTP server does not allow the HEAD request method. In cases like that, we would have to resort to GET requests. I will leave that as an exercise for you.

Working with News Feeds

News feeds or RSS feeds have become quite popular in the past several years. Initially conceived as a means to publish web site update notifications, they have quickly become the preferred way to publish any type of updates from any program. People have been using them to announce new posts on personal web logs (blogs) or Internet forums, new files in download sites, and all sorts of updates on all types of objects. Looking at the RSS acronym used everywhere, one may think that there is a well defined and widely accepted standard for publishing those news items. Sadly, this is not the case. The RSS acronym has been used to mean different things, and it has several versions and variations as a format, each having its own advantages and disadvantages. Looking at the different formats, I am tempted to say that the only thing in common among all of them is that they are all based on XML.

There are two major branches of RSS file formats: one is based on the Resource Description Framework (RDF) recommendation; the other is not. Here are the formats that are based on the RDF recommendation:

- *RSS 0.90*: Developed by Netscape, this acronym stood for RDF Site Summary. The format was based on an early draft of the RDF recommendation.
- *RSS 1.0*: Again standing for RDF Site Summary, this is an open format, developed and based on the final RDF recommendation. As such, it is not compatible with RSS 0.90.
- *RSS 1.1*: This is another open format that was meant to replace RSS 1.0. It was not endorsed by any major organization or group and is not very popular.

The other important branch is the RSS 2.* branch. Those formats are not based on RDF and are not compatible with the preceding ones:

- *RSS 0.91*: This simplified format released by Netscape is not based on RDF, and the acronym stands for Rich Site Summary. This is one of the most popular formats to date.
- *RSS 0.92 through 0.94*: Those are extensions to the RSS 0.91 format, which are mostly compatible with it.
- *RSS 2.0.1*: This is an update released shortly after RSS 2.0. It is mostly compatible with the other formats in this branch, and the only major addition is the explicit extensibility support that was added. At this point in time, the acronym stands for Really Simple Syndication.

There is another feed format that has grown in popularity: Atom. It is an attempt at getting a clean start at defining a better feed format. It has been standardized and adopted by large organizations, such as Google.

I can see that your head is starting to spin from all those formats. In this chapter, we will try to support one RSS format from the two branches: RSS 1.0 and RSS 2.0. In addition, we will support Atom feeds. Being able to support the three most popular formats will enable us to write scripts that can consume updates published by other programs and react to various events.

Getting Feeds

Let's start with the Atom format. We will need to download the XML content of the feed document and will need to extract relevant information. PowerShell's has excellent support for XML data so that should not be a problem. Let's now create a script called `Get-RssFeed.ps1` that we will gradually evolve. In the beginning, we will need to get a list of updates and their titles and message bodies. We will use the PowerShell team blog's Atom feed for our tests. Here is the code:

```
param ($url)

$client = New-Object System.Net.WebClient

$contents = $client.DownloadString($url)
$feed = [xml] $contents

function Get-AtomPosts($feed)
{
    $feed.feed.entry | foreach {
        @{ "Title" = $_.title; "Body" = $_.content.$("#text") }
    }
}

Get-AtomPosts $feed
```

We get the XML contents as a string and convert them to an XML document. We then go over the Atom document structure to create a list of Hashtable objects, each of them having a Title and Body values. The Atom XML format has a root `<feed>` element that has a collection of `<entry>` objects. Each entry has a `<title>` element and a `<content>` element. The latter's inner text holds the entry body. Let's run the code and see what happens:

```
PS> .\Get-RssFeed.ps1 http://blogs.msdn.com/powershell/atom.xml
```

Name	Value
----	----
Body	<P>Next week we'll be releasing a Co...
Title	CTP: Watch This Space
Body	<p>Next week we'll be releasing a Co...
Title	CTP: Versioning
...	

I know what you are thinking—the <P> tags mean that the entry bodies contain HTML text. We can use that fact to format entries and display them in a browser. Let's try it with the first entry:

```
PS> $url = "http://blogs.msdn.com/powershell/atom.xml"
PS> $entries = .\Get-RssFeed.ps1 $url
PS> $entry = $entries[0]
PS> $html = @"
<html>
<head>
<title>$($entry.Title)</title>
</head>
<body>
$($entry.Body)
</body>
</html>
"@
PS> Set-Content -path entry.html -value $html
PS> Invoke-Item entry.html
```

We generated a small HTML document and opened it in the browser. Figure 17-7 shows how that looks like in Mozilla Firefox.

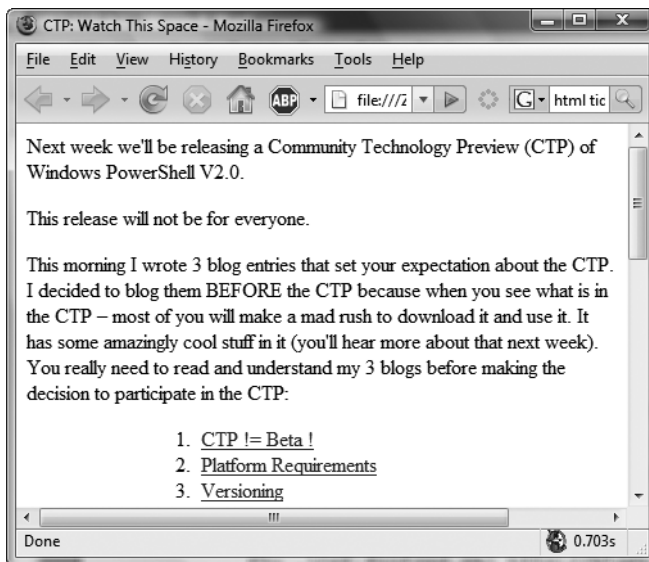


Figure 17-7. Mozilla Firefox displaying our first feed entry

Let's now move on to adding RSS 2.0 support. The logic is almost the same, with the only difference being that we have to account for the different structure that RSS 2.0 documents have. The root element for our XML document is now <rss>. It usually contains one <channel> element, which in turn, contains many <item> elements. The <item> elements are our entries—we need their title and description properties. Here is the code:

```
param ($url)

$client = New-Object System.Net.WebClient

$contents = $client.DownloadString($url)
$feed = [xml] $contents

function Get-RssPosts($feed)
{
    $feed.rss.channel.item | foreach {
        @{ "Title" = $_.title; "Body" = $_.description }
    }
}

Get-RssPosts $feed
```

It is a good thing that the server software that PowerShell team blog runs on supports RSS 2.0 feeds too. We will use that one in our experiments. Here is what happens when we call our script:

```
PS> .\Get-RssFeed.ps1 http://blogs.msdn.com/powershell/rss.xml
```

Name	Value
----	-----
Body	<P>Next week we'll be releasing a Co...
Title	CTP: Watch This Space
Body	<p>Next week we'll be releasing a Co...
Title	CTP: Versioning

We want to generate a collection of Hashtable objects, each having a Title and Body properties just like we did with the Atom entries.

Finally, let's handle the RDF branch of the RSS formats. We will use the Slashdot.org RSS feed; it uses the RSS 1.0 format. The RDF formats have a root <RDF> element that contains a number of <item> child elements. We are interested in each item's title and description properties. Here is the code that handles that format:

```

param ($url)

$client = New-Object System.Net.WebClient

$content = $client.DownloadString($url)
$feed = [xml] $content

function Get-RdfPosts($feed)
{
    $feed.RDF.item | foreach {
        @{ "Title" = $_.title; "Body" = $_.description }
    }
}

Get-RdfPosts $feed

```

Here is what we get after running the script:

```
PS> .\Get-RssFeed.ps1 http://rss.slashdot.org/Slashdot/slashdot
```

Name	Value
Body	ThinkingInBinary writes "The results...
Title	Carnegie Mellon Wins Urban Challenge
Body	einhverfr writes "According to an ar...
Title	Ultracapacitors Soon to Replace Many...

We now have three versions of the same script that fetch a feed document, process it, and emit a collection of feed entries in a unified format. We can now try to merge the three versions into one. To do that, we need some code that will check the structure of the XML document that we got from the server. According to the structure, the code will process the document using `Get-AtomPosts`, `Get-RssPosts`, or `Get-RdfPosts`. We will check the document root element to determine what file format we have. Here is the final version of the code:

```

param ($url)

$client = New-Object System.Net.WebClient

$content = $client.DownloadString($url)
$feed = [xml] $content

function Get-RdfPosts($feed)
{
    $feed.RDF.item | foreach {
        @{ "Title" = $_.title; "Body" = $_.description }
    }
}

```

```

function Get-AtomPosts($feed)
{
    $feed.feed.entry | foreach {
        @{ "Title" = $_.title; "Body" = $_.content.$("#text") }
    }
}

function Get-RssPosts($feed)
{
    $feed.rss.channel.item | foreach {
        @{ "Title" = $_.title; "Body" = $_.description }
    }
}

if ($feed.feed -ne $null)
{
    Get-AtomPosts $feed
}
elseif ($feed.rss -ne $null)
{
    Get-RssPosts $feed
}
elseif ($feed.RDF -ne $null)
{
    Get-RdfPosts $feed
}
else
{
    throw "Unsupported feed format"
}

```

The code tries several strategies and throws an exception if it finds nothing suitable. We can now run this script with any of the preceding feed formats without having to remember which URL corresponds to which format:

```
PS> .\Get-RssFeed.ps1 http://blogs.msdn.com/powershell/atom.xml
```

Name	Value
----	-----
Body	<P>Next week we'll be releasing a Co...
Title	CTP: Watch This Space
...	

```
PS> .\Get-RssFeed.ps1 http://blogs.msdn.com/powershell/rss.xml
```

Name	Value
----	-----
Body	<P>Next week we'll be releasing a Co...
Title	CTP: Watch This Space
...	

```
PS> .\Get-RssFeed.ps1 http://rss.slashdot.org/Slashdot/slashdot
```

Name	Value
----	-----
Body	ThinkingInBinary writes "The results..."
Title	Carnegie Mellon Wins Urban Challenge
...	

Calling Web Services

Web services, or also called XML web services, have been a hot topic for several years now. They are in essence a remoting mechanism, a system designed to enable machine-to-machine communication. Based on an XML format for exchanging messages, the service invocation effectively represents a method call. The remote object and its target method are fully qualified with a URL and parameters, and results are transported using XML messages, typically over HTTP.

So, why are web services important to us? Similar to RSS feeds, they are a widely used method of exchanging data between systems. Many applications and services expose information using web services, and we can use web services to get data from remote servers, send data to those servers, or issue commands that will affect their behavior. The best thing about web services is that they are completely implementation agnostic, and we can use them as a way to connect heterogeneous systems implemented using different technologies. In this chapter, we will focus on web services implemented with Microsoft .NET, but the same approach should work for services implemented using other technologies such as Java, PHP, and so on.

Calling Web Services Using HTTP GET Requests

Usually, web services are not called using raw HTTP requests; many programming environments analyze the Web Service Description Language (WSDL) manifest associated with the web service and generate local proxy objects that mimic the remote object's methods. Those proxies take care of transporting the parameters and return values to the client code. We can certainly use .NET's tools and generate proxy objects that can allow us to work in a similar manner. The drawback to that approach, though, is that you have to either generate code and compile temporary assemblies or use some serious .NET programming black magic to generate in-memory assemblies that contain the proxy objects. There is a simpler way—PowerShell makes handling XML so easy that we can craft an HTTP request with `System.Net.WebClient` and parse the XML that we get back from the remote service. The code that we need to process the XML result is about two lines long.

To demonstrate how to call a web service using a HTTP GET request, we will get the weather forecast from `http://www.webservice.net/WeatherForecast.asmx`. We will use the `GetWeatherByPlaceName` remote method. To pass the `PlaceName` parameter, we will need to append a query string parameter to the URL—`?PlaceName=<CityName>`. We will save our code to

a script file named `Call-WebServiceGet.ps1`. My favorite place for this example is New York. Here is the code that initiates the request and processes the result:

```
param ($Proxy)

$serviceUrl = "http://www.webservices.net/WeatherForecast.asmx"
$city = "New York"
$url = "$serviceUrl/GetWeatherByPlaceName?PlaceName=$city"

$client = New-Object System.Net.WebClient

if ($Proxy)
{
    $proxyConfig = New-Object System.Net.WebProxy -arg $Proxy
    $client.Proxy = $proxyConfig
}

$contents = $client.DownloadString($url)
$xmlDoc = [xml] $contents
$xmlDoc.WeatherForecasts.Details | `
    foreach {
        $_.WeatherData | select Day,MaxTemperatureC,MaxTemperatureF
    }
```

The code assembles a URL that contains the `?PlaceName=New York` query string parameter. We convert the resulting string to an XML document. That document has a root node of `<WeatherForecasts>`. We are interested in the forecast for the next seven days that is contained by the `<Details>` child node. It is represented by a number of `<WeatherData>` objects stored inside. We pipe the `WeatherData` objects through the `select` cmdlet to get a formatted report that contains the day and the maximum temperatures in Celsius and Fahrenheit. For diagnostics purposes, we have added support for a proxy server, so that we can pipe the request through Fiddler and see what goes on under the hood. Here is what we get after running our script:

```
PS> .\Call-WebServiceGet.ps1 -Proxy http://localhost:8888
```

Day	MaxTemperatureC	MaxTemperatureF
Monday, November 05,...	15	59
Tuesday, November 06...	13	56
Wednesday, November ...	11	52
Thursday, November 0...	8	47
Friday, November 09,...	8	47
Saturday, November 1...	10	50
Sunday, November 11,...	12	53

Figure 17-8 shows Fiddler's view of the request. The Session Inspector tab shows the raw request and the XML view of the response.

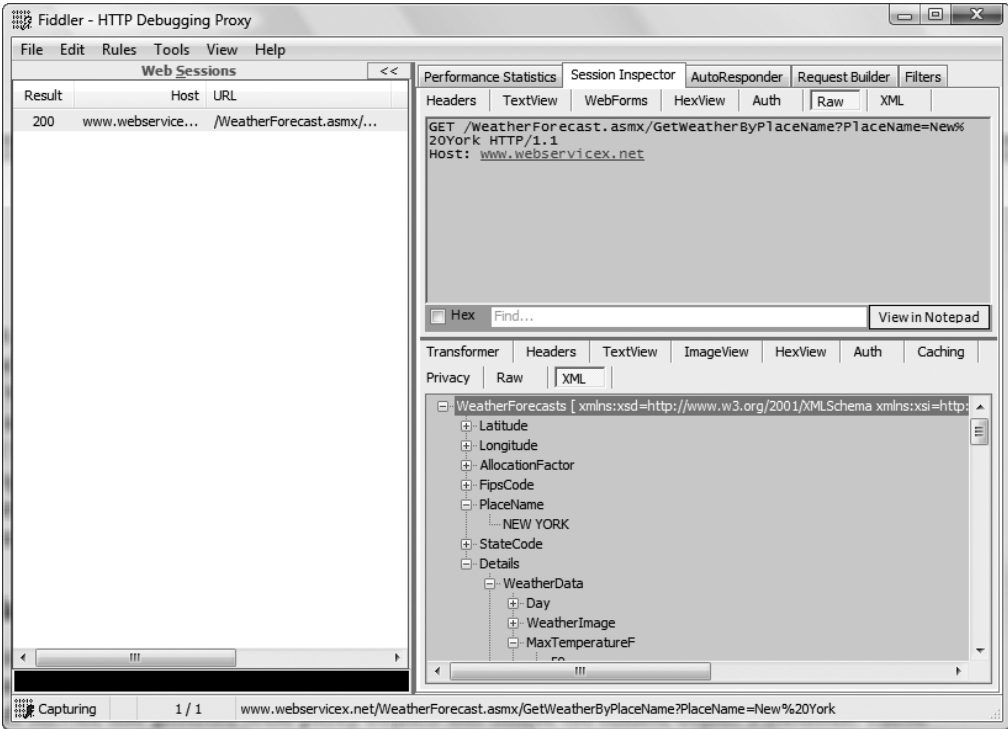


Figure 17-8. Fiddler showing a HTTP GET request to a web service

Calling Web Services Using HTTP POST Requests

The GET HTTP method is the easiest to use, but the thing that makes it easiest to use is also its biggest weakness. The parameters are sent in the query string part of the URL, and that space has limits. The World Wide Web Consortium (W3C) documents suggest that user agents limit URLs and query string parameters to 2KB of data. We have to use a different method of invoking a web service if we need to provide larger amounts of data as the input parameter. HTTP POST parameters are again represented as `name=value` strings similar to query string parameters. The difference from a GET request is that those parameters are sent in the request body and the maximum size is limited to the physical capacity of the server and the network. We do not need to encode parameters ourselves; this is fully supported by `System.Net.WebClient`. The only thing we have to do is stuff the parameters in a `NameValueCollection` object and pass that to the `UploadValues` method. A `NameValueCollection` is really a dictionary, similar to a `Hashtable` that has strings as its keys and values. We will create a new script, `Call-WebServicePost.ps1`, that will get the weather forecast for the next week for New York, this time using a POST request. Here is the code:


```

param ($Proxy)

$serviceUrl = "http://www.webservices.net/WeatherForecast.asmx"
$url = "$serviceUrl/GetWeatherByPlaceName"

$values = New-Object Collections.Specialized.NameValueCollection
$values["PlaceName"] = "New York"

$client = New-Object System.Net.WebClient

if ($Proxy)
{
    $proxyConfig = New-Object System.Net.WebProxy -arg $Proxy
    $client.Proxy = $proxyConfig
}

$rawData = $client.UploadValues($url, $values)

$contents = [Text.Encoding]::UTF8.GetString($rawData)
$xmlDoc = [xml] $contents
$xmlDoc.WeatherForecasts.Details | `
    foreach {
        $_.WeatherData | select Day,MaxTemperatureC,MaxTemperatureF
    }

```

This time, we add the `PlaceName` parameter to the `$values` object and call the `UploadValues` method. By default, `UploadValues` initiates a POST request, so we do not need to explicitly state that. There is a slight inconvenience related to using `UploadValues`; it returns an array of bytes. We store those bytes in the `$rawData` variable, and we have to resort to the UTF8 encoding object to turn those bytes into a string. The rest of the code is absolutely the same as in our GET example; the returned XML has absolutely the same format. As you would expect, the result from running the script looks pretty similar:

```
PS> .\Call-WebServicePost.ps1 -Proxy http://localhost:8888
```

Day	MaxTemperatureC	MaxTemperatureF
Monday, November 05,... 15	59	
Tuesday, November 06... 13	56	
Wednesday, November ... 11	52	
Thursday, November 0... 8	47	
Friday, November 09,... 8	47	
Saturday, November 1... 10	50	

Figure 17-9 shows Fiddler's view of the request. Note that the parameters are now transmitted in the request body, and the resulting XML is the same.

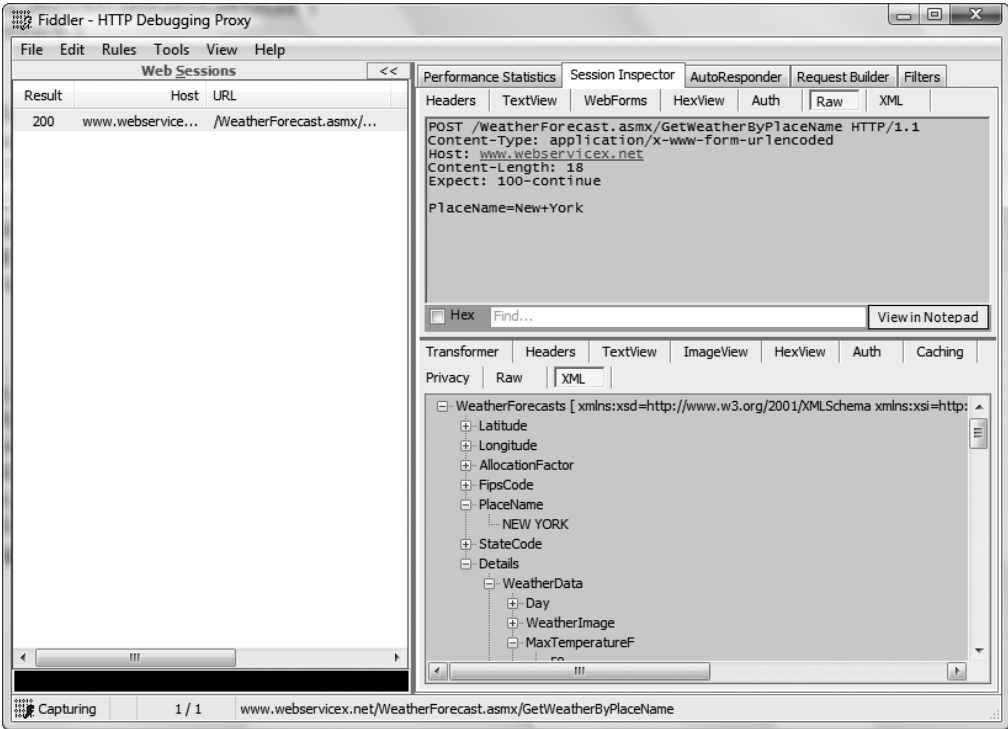


Figure 17-9. Fiddler showing an HTTP POST request to a web service

Calling Web Services Using the SOAP Protocol

SOAP stands for Simple Object Access Protocol. It is a specification for representing web service method calls and object method calls in general as XML messages. The SOAP protocol is transport agnostic. Although XML messages are typically transported in HTTP POST requests, we are certainly not limited to that. There are projects that can transport a SOAP message over SMTP in a MIME-encoded e-mail message. We will not go that route, however; we will focus on SOAP messages over HTTP POST requests.

First, why do we need to bother with another way to format a POST request when we already have one? The POST method that we just looked at is far simpler to use than SOAP. The problem with the GET and POST methods you have seen so far is that they are not standardized. They are Microsoft extensions to calling web services and may not be available for web services implemented using non-Microsoft technologies. In addition, the HTTP-GET and HTTP-POST methods for calling a service are disabled by default; they may not be enabled for the service we are trying to call. SOAP, even though it is the most complex of the three methods, is the most widely supported protocol. It will work for services where the other methods may fail.

Let's see what a sample SOAP message looks like. Here is the SOAP message that we can use to get the weather forecast for New York using our sample web services:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetWeatherByPlaceName xmlns="http://www.websvcicex.net">
      <PlaceName>New York</PlaceName>
    </GetWeatherByPlaceName>
  </soap:Body>
</soap:Envelope>
```

The preceding code may seem daunting, but most of it is just boilerplate code that we can copy and paste without digging much deeper. According to the specification, we need to have a <soap:Envelope> element with a <soap:Body> one inside. We then have to include an element with the name of the web service method that should contain elements containing each of the parameter values. In addition to sending the SOAP message in the request body, we need to send a SOAPAction request header with a value containing the URL of our web service and the method we are calling. Another necessary header is the Content-Type one; we need to indicate to the server that we are submitting XML data encoded in UTF-8. We will call our weather forecast service using SOAP from a new script called Call-WebServiceSoap.ps1. That script will use the System.Net.WebClient UploadString method to post the SOAP message to the server. Here is the code:

```
param ($Proxy)

$serviceUrl = "http://www.websvcicex.net/WeatherForecast.asmx"

$city = "New York"

$soapMessage = @"
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetWeatherByPlaceName xmlns="http://www.websvcicex.net">
      <PlaceName>$city</PlaceName>
    </GetWeatherByPlaceName>
  </soap:Body>
</soap:Envelope>
"@

$client = New-Object System.Net.WebClient
```

```

if ($Proxy)
{
    $proxyConfig = New-Object System.Net.WebProxy -arg $Proxy
    $client.Proxy = $proxyConfig
}

$client.Headers.Add("SOAPAction", `
    "http://www.webservice.net/GetWeatherByPlaceName")
$client.Headers.Add("Content-Type", "text/xml; charset=utf-8")

$content = $client.UploadString($serviceUrl, $soapMessage)

$xmlDoc = [xml] $content

$response = $xmlDoc.Envelope.Body.GetWeatherByPlaceNameResponse
$response.GetWeatherByPlaceNameResult.Details | `
    foreach {
        $_.WeatherData | select Day,MaxTemperatureC,MaxTemperatureF
    }

```

The script is certainly longer, but apart from the long string that contains the SOAP message, the code does pretty much the same thing as in the scripts you have seen so far. We are using a double-quoted string to generate the SOAP message, so that we can easily inject the `$city` variable value inside. The request has to carry two important headers: `SOAPAction` and `Content-Type`. The former tells the server which web service method to invoke, and the latter specifies that the message body contains XML text encoded using UTF-8. The resulting XML is a bit different than our previous results too; it contains another `<soap:Envelope>` with a `<soap:Body>` element inside. The actual response is wrapped in a `<GetWeatherByPlaceNameResponse>` element. Once we get to that element, we can get to the `<Details>` element inside that holds the familiar `WeatherData` objects. Here is the result we get after running our script:

```
PS> .\Call-WebServiceSoap.ps1 -Proxy http://localhost:8888
```

Day	MaxTemperatureC	MaxTemperatureF
---	-----	-----
Monday, November 05,...	15	59
Tuesday, November 06...	13	56
Wednesday, November ...	11	52
Thursday, November 0...	8	47
Friday, November 09,...	8	47
Saturday, November 1...	10	50

We are getting the same property values and formatting the data in the same way, so there is no surprise that the output looks similar to what we have seen before. Figure 17-10 shows the request and response messages in Fiddler's XML view.

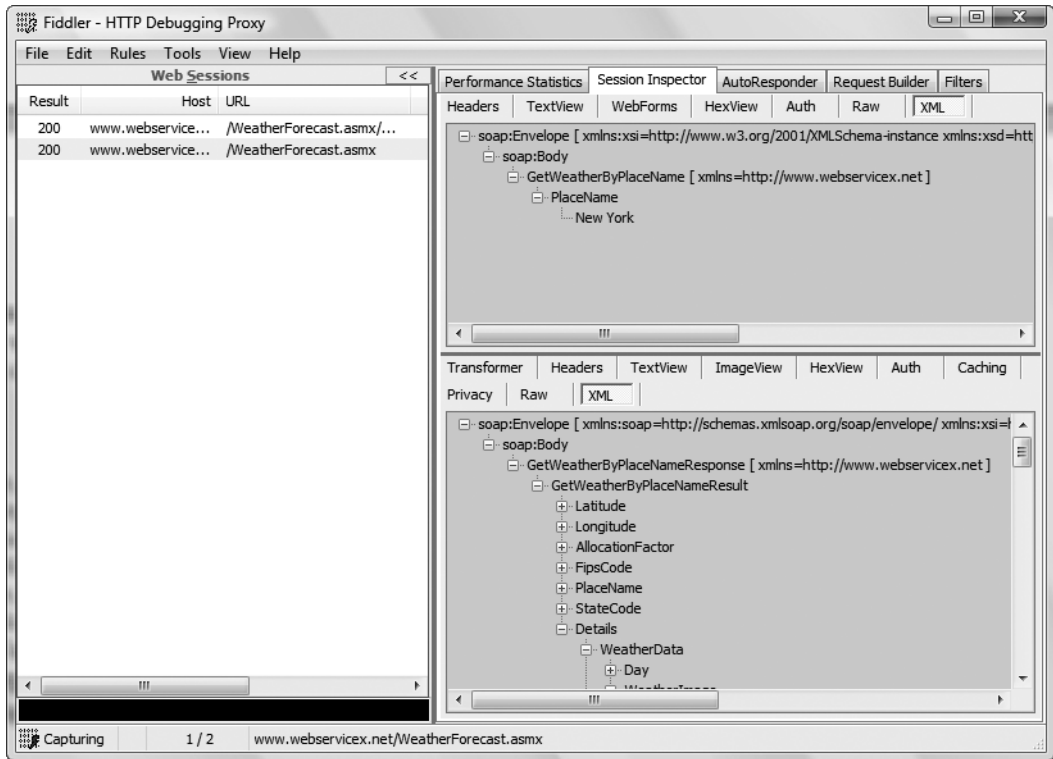


Figure 17-10. Fiddler showing the request and response SOAP messages involved in calling a web service

Handling FTP Transactions

It is a little known fact that `System.Net.WebClient` can handle FTP connections too. Most people mean HTTP transactions when they refer to the Web, and in a sense, that makes the `WebClient` name a bit misleading. The FTP support this class offers is limited in that it only supports passive FTP connection. Passive FTP mode is the default for many clients; it means that the client connects to the server and is then given a new address and port to connect to. Active mode, where the server connects back to the client, is not supported by the `WebClient` class. FTP is an ancient protocol, and it has been long known to have some security issues. For one thing, the credentials are sent with no encryption over the network, and that makes us vulnerable to traffic sniffing. When possible, avoid this protocol and replace it with one of the two secure FTP protocols: Secure FTP (FTP sessions running over encrypted SSH channels) or SSH File Transfer Protocol, both of which have built-in encryption.

Downloading Files from an FTP Server

Downloading a file from an FTP server using the `WebClient` class looks like downloading a file from a password-protected HTTP server. We have to set up a `NetworkCredential` object with our user name and password and call the `DownloadFile` method. We can use `DownloadString`

too, but let's assume here that we want to automatically save the file to a new location. Let's create a simple script called `Get-Ftp.ps1` that will accept four parameters: the URL for our target file, a local file, user name, and password. Here is the code:

```
param ($url, $file, $userName, $password)

$client = New-Object System.Net.WebClient

$credentials = New-Object System.Net.NetworkCredential `
    -arg $userName, $password
$client.Credentials = $credentials

$client.DownloadFile($url, $file)
```

We use this by providing an `ftp://` URL and an absolute local path. We form an absolute path that points to the current folder by using the `Join-Path` and `Get-Location` cmdlets:

```
PS> .\Get-Ftp.ps1 -url ftp://deshev.com/test.txt `
-file (Join-Path (Get-Location) test.txt) `
-userName ftpptest -password testpass
```

```
PS> type test.txt
Test file downloaded from FTP.
```

Uploading Files to an FTP Server

Again, uploading FTP files is similar to working with HTTP. Uploading a file to a remote FTP server using `WebClient` looks exactly like uploading a file using an HTTP POST request. We can read the entire file contents and pass that to the `UploadString` method. The better alternative is to call the `UploadFile` method and pass the file name. We will do that in a script called `Put-Ftp.ps1` that accepts four parameters: the URL of the uploaded file, the local file name, user name and password. Here is the code that does the actual uploading:

```
param ($url, $file, $userName, $password)

$client = New-Object System.Net.WebClient

$credentials = New-Object System.Net.NetworkCredential `
    -arg $userName, $password
$client.Credentials = $credentials

$client.UploadFile($url, $file)
```

Again, we call the script by pointing to our test FTP server and passing the absolute path to the local file that we want uploaded. Let's get the `Get-Ftp.ps1` file and upload it to the server:

```
PS> .\Put-Ftp.ps1 -url ftp://deshev.com/Get-Ftp.ps1 `
-file (gi Get-Ftp.ps1).FullName `
-user ftpptest -password testpass
```

```
PS>
```

We do not get an error, and everything seems to be fine. Figure 17-11 shows the uploaded file on our FTP server.

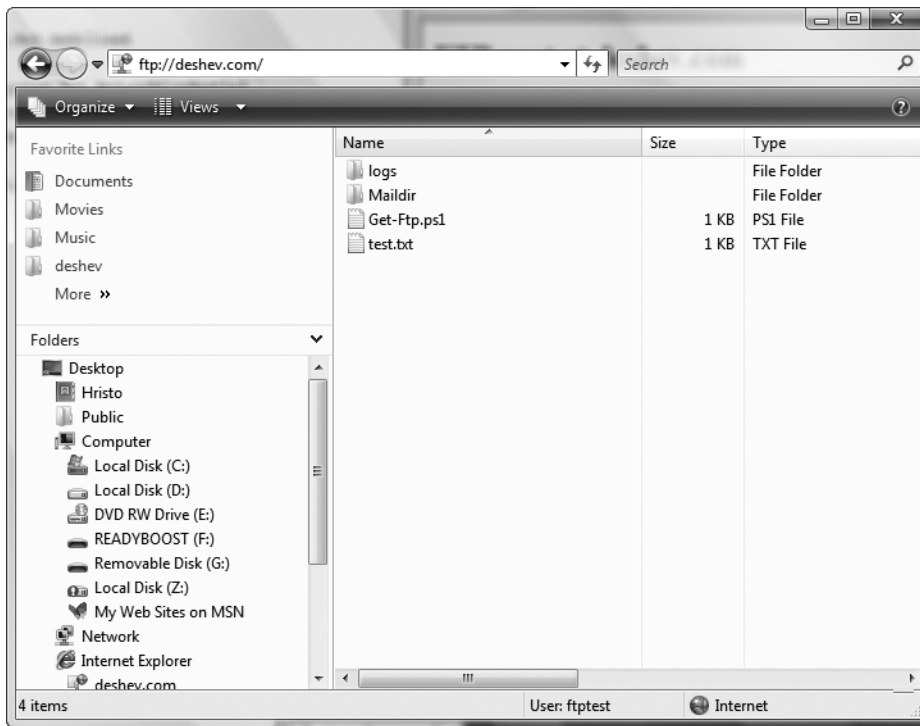


Figure 17-11. *The Windows Vista Explorer window showing the contents of the FTP server root folder with our newly uploaded file*

Summary

Maybe I should have named this chapter “The Cult of System.Net.WebClient.” We can do almost everything related to the Web or to FTP transfers with this class. Most often, I need simple file download functionality from a web or FTP server. I need file uploads and HTTP POST requests relatively often, yet those are significantly rarer than file downloads. We covered a lot of advanced scenarios in this chapter: processing HTML and XML is not something that you see every day. Still, the Web hosts tons of different HTML- and XML-based files that you may encounter. When you do, you will know how to process them with PowerShell. I cannot stress enough the sheer power behind PowerShell’s XML processing facilities; several lines of code are enough to process and extract data from an RSS feed or a SOAP message. I am sure that this will hold true for other XML-based file formats.

Probably the most important thing you can take away from this chapter is the Fiddler tool. This little program is indispensable for anyone working with HTTP. In addition to helping us troubleshoot problems, it can help a lot with automating a task that you know how to perform manually. Want to automate an action you do routinely on a web site? Configure your browser’s proxy settings so that its requests go through Fiddler and see what gets sent to the server. Then, you need to just send the same data and headers from your PowerShell script to automate the task.



Sending E-mail

All types of system automation will sooner or later have to deal with e-mail. Long running processes often call home when they are finished. Complex operations complete by sending their execution log to a central place. Failures and special conditions that require user actions usually distribute notifications via e-mail to all interested parties. Unlike some automation environments, PowerShell does not provide a built-in facility that sends e-mail. This may seem like a big omission at first, but things get a lot better when we examine the alternatives. The Microsoft.NET framework 2.0 already provides excellent support for sending SMTP mail with its `System.Net.Mail` classes, and we will use those to build a script that can do the job for us. PowerShell Community Extensions, a popular open source project, also provides a cmdlet for sending mail that is built on top of `System.Net.Mail`. I will cover the PowerShell Community Extensions in greater detail in Chapter 21.

Why write our own script instead of just using the existing tool? Scripting .NET objects and sending mail on our own has several advantages over using an extension cmdlet: it provides us with greater flexibility and can cater to our specific needs, and it frees us from the requirement of having to install an extension that is usually a hassle and might require administrator privileges on the target machine.

Using System.Net.Mail

`System.Net.Mail` is a namespace in the .NET 2.0 Framework that provides classes dealing with all the intricacies involved with sending e-mail. Those classes know how to create a mail message, how to encode it and its attachments, and how to talk the server into sending it to the correct recipients. The basic mail sending functionality is quite simple to achieve: we create a `MailMessage` object; set it up with the correct recipient address, subject, and body; and pass it to an `SmtpClient` that actually sends the message over the wire. The following sample script, `send-plain.ps1`, does exactly that:

```
#create the mail message
$mail = New-Object System.Net.Mail.MailMessage

#set the addresses
$mail.From = New-Object System.Net.Mail.MailAddress("test@deshev.com")
$mail.To.Add("test@deshev.com")
```

```
#set the content
$mail.Subject = "Hello PowerShell";
$mail.Body = "Sending mail is easy!";

#send the message
$smtp = New-Object System.Net.Mail.SmtpClient -argumentList "mail.deshev.com"
$smtp.Credentials = New-Object System.Net.NetworkCredential `
    -argumentList "test@deshev.com","testpass"
$smtp.Send($mail)
```

Note that we can even spoof the From address if we want to. Please do not abuse that! The `SmtpClient` object requires some information about our mail server. We pass the server address to the constructor, and then we provide a new `NetworkCredential` object that contains our user name and password. Note that we are using dummy values for the mail address, the mail server configuration, and the login credentials. You will have to obtain the correct settings for your mail server and SMTP account.

Keeping all sorts of configuration scattered in various scripts is not a good idea, as we have to remember what goes where and change many files when something in our configuration changes—not to mention that it is a security risk to embed credentials in plain text files. We will move those settings to a better location later. A useful thing to know is that `SmtpClient` supports SSL authentication and even works with Google's free Gmail service. For all practical needs, we do not need better SMTP support than that!

Building a Reusable Script

The script in the previous section does a fine job of sending a simple message, but it is not reusable at all. Should we have to send a message with a different recipient, we would have to copy the code, paste it to a new script, and make the change there. That leaves us with two almost identical scripts that are twice as hard to maintain. We need to move the mail server configuration and the user credentials to a separate place and include them in our scripts. Additionally, we have to abstract the message data, the recipient, subject, and body and introduce parameters that allow us to pass different values from the command line. Let's start with the configuration.

A slightly more secure location for storing configuration data is the PowerShell user profile script. It is stored in the Documents folder on Windows Vista and My Documents on Windows XP and earlier. The profile script resides inside the `WindowsPowerShell` subfolder and is named `profile.ps1` or `Microsoft.PowerShell_profile.ps1`. Create a file with one of those names in that location if you do not have one already. Which one of the two names should you pick? It does not matter—PowerShell will execute both scripts if it finds them. We will move the mail server configuration there:

```
#mail server configuration
$smtpServer = "mail.deshev.com"
$smtpUser = "test@deshev.com"
$smtpPassword = "testpass"
```

After restarting PowerShell, the three variables—`$smtpServer`, `$smtpUser`, and `$smtpPassword`—will always be available to our scripts. Since we have those variables declared in the profile

script, we can now delete their declaration from the original script. For an in-depth description of the PowerShell profile scripts, please refer to Chapter 11.

To separate our message data from our script and introduce parameters, we introduce two functions: `Create-Message`, which will configure a `MailMessage` object, and `Send-Message`, which will work with an `SmtpClient` object. `Create-Message` takes the recipient address, message subject, and message body as parameters and returns a `MailMessage` object:

```
function Create-Message([string] $to = $(throw "'to' cannot be null"),
    [string] $subject = "(no subject)",
    [string] $body = $(throw "'body' cannot be null"))
{
    $message = New-Object System.Net.Mail.MailMessage
    $message.From = New-Object System.Net.Mail.MailAddress `
        ("test@deshev.com")

    #configure recipients
    $message.To.Add($to)

    $message.Subject = $subject
    $message.Body = $body

    return $message
}
```

We require that callers always pass the `$to` and `$body` parameters and default to a “(no subject)” line if the `$subject` parameter is missing.

The `Send-Message` function takes a single parameter, a preconfigured `MailMessage` object:

```
function Send-Message([System.Net.Mail.MailMessage] $message)
{
    $smtp = New-Object System.Net.Mail.SmtpClient -argumentList $smtpServer
    $smtp.Credentials = New-Object System.Net.NetworkCredential -argumentList `
        $smtpUser,$smtpPassword
    $smtp.Send($message)
}
```

It is best that we move those functions to a separate file and include it as a function library in scripts that we execute. We will call the new file `mailutils-plain.ps1` and include it with the dot-sourcing notation like this:

```
.. \mailutils-plain.ps1
```

Our driver script, `send-mail-plain.ps1`, now looks like this:

```
param([string] $to = $(throw "'to' cannot be null"),
    [string] $subject = "(no subject)",
    [string] $body = $(throw "'body' cannot be null"))

.. \mailutils-plain.ps1
```

```
$message = Create-Message $to $subject $body
```

```
Send-Message $message
```

All it does is get parameters from the command line, pass them to `Create-Message`, and call `Send-Message` afterward. Here is how we use our script:

```
.\send-mail-plain.ps1 -to test@deshev.com -subject "Look, Ma" `
    -body "I'm a reusable script!"
```

Configuring Recipients

Our current solution has the limitation of being able to send the message to only one recipient. Of course, we can loop or pipe a collection through `ForEach-Object` and call the script multiple times, but that is quite inefficient, as it would always open a new connection to the server, send the mail, and then close it just to open it back on the next iteration. Performance is not the only problem: there is no way for us to set up a CC or BCC list for our carbon copy recipients. In addition, we often want to use a friendly name for our recipients, so that users see a real name like “Mike Smith | ACME Corp.” instead of a plain “msmith@acme.com.”

Adding multiple recipients requires that we pass a collection of addresses as a parameter to our script. We then have to change the `Create-Message` function and make it go over that collection. The easiest way to do that is to use `foreach`:

```
$to | foreach { $message.To.Add($_) }
```

`foreach` is nice enough to work with a single item as if it were passed a collection with one item, so that we do not have to pass an array when we need to send a message to one recipient only.

Configuring the CC and BCC recipients for a message works in the same way as configuring for To recipients does: we have to use the `MailMessage` class’s CC and BCC properties. Again, we use `foreach` to allow for multiple recipients:

```
$cc | foreach { $message.CC.Add($_) }
```

```
$bcc | foreach { $message.BCC.Add($_) }
```

Our `Create-Message` function now becomes this:

```
function Create-Message($to = $(throw "'to' cannot be null"),
    $cc = @(),
    $bcc = @(),
    [string] $subject = "(no subject)",
    [string] $body = $(throw "'body' cannot be null"))
{
    $message = New-Object System.Net.Mail.MailMessage
    $message.From = New-Object System.Net.Mail.MailAddress("test@deshev.com")

    #configure recipients

    $to | foreach { $message.To.Add($_) }
    $cc | foreach { $message.CC.Add($_) }
    $bcc | foreach { $message.BCC.Add($_) }
```

```

$message.Subject = $subject
$message.Body = $body
return $message
}

```

Note that we default to empty collections for the `$cc` and `$bcc` parameters, so that the user is not required to provide them. The updated `Create-Message` function resides in a new version of our script library: `mailutils-multiple-recipients.ps1`. We also need to change the driver script to add the `$cc` and `$bcc` parameters and pass them to `Create-Message`:

```

param($to = $(throw "'to' cannot be null"),
      $cc = @(),
      $bcc = @(),
      [string] $subject = "(no subject)",
      [string] $body = "")

. .\mailutils-multiple-recipients.ps1

$message = Create-Message $to $cc $bcc $subject $body
Send-Message $message

```

Note that the driver script now includes the new version of our script library. We name the driver script in a similar fashion, `send-mail-multiple-recipients.ps1`. Here is how we can use our script to send a message to many people as direct, carbon copy, or blind carbon copy recipients:

```

.\send-mail-multiple-recipients.ps1 -to @("test@deshev.com", `
    "test@deshev.com") -cc @("test@deshev.com", "test@deshev.com") `
    -bcc @("hristo@deshev.com", "test@deshev.com") `
    -subject 'hi everyone' -body 'test'

```

Caution Blind copies are not actually blind. Normally, a mail recipient should have no way of telling if that message has been blind carbon copied to another person. `System.Net.Mail`, unfortunately, has a bug that will allow a recipient to do that. Messages will carry an `X-Receiver` header that will list all recipients, and that header will be visible to everyone on the `To`, `CC`, and `BCC` lists. Mail clients do not typically display those headers and make them visible right away, but anyone who bothers to view the message headers will get the full recipient list.

We can provide a recipient's e-mail address that contains a friendly name besides the actual address by using a `MailAddress` object and pass that to our script as the `$to`, `$cc`, or `$bcc` parameter. PowerShell's dynamic nature will accept that object, and our script will still work. Going through `New-Object` and creating a new `MailAddress` with code like this

```

$addr = New-Object System.Net.Mail.MailAddress `
    -arg @("test@deshev.com", "Test User3")

```

can be quite tedious. Fortunately, we can provide a properly formatted address string like “Mike Smith | ACME Corp <msmith@acme.com>”, and System.Net.Mail will deal with it properly. Our script needs no changes to support that:

```
.\send-mail-multiple-recipients.ps1 -to "Test User2 <test@deshev.com>" `
    -subject 'friendly name'-body 'test'
```

E-mail clients will display the message using both the friendly name and the actual address. Figure 18-1 shows how our message looks in Windows Mail, the mail client bundled with Windows Vista.

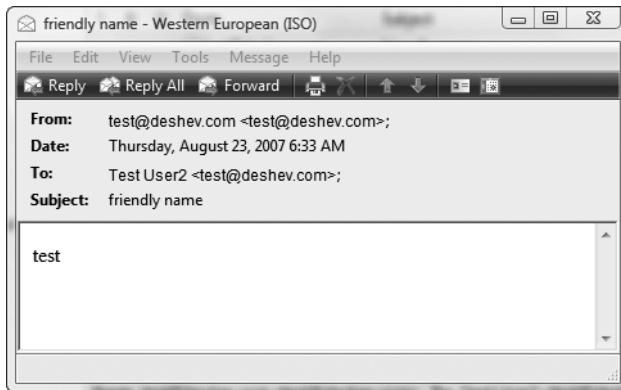


Figure 18-1. Windows Mail showing our recipient's display name, Test User2

Figure 18-2 shows how the same message looks in Outlook 2007.

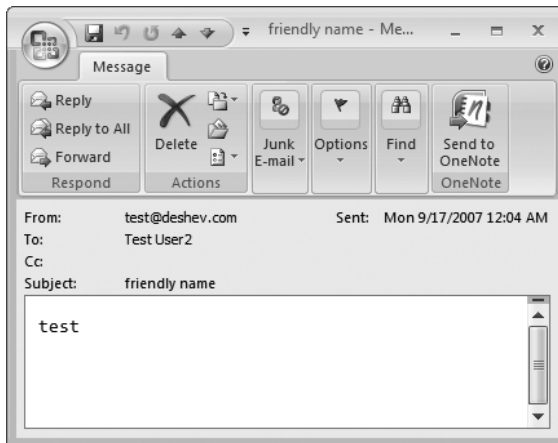


Figure 18-2. Outlook 2007 showing our recipient's display name, Test User2

Working with Message Headers

Frequently, we need to set our message with additional headers that will signal the priority, Reply-To address, and many other things that we, our mail client, or our mail server can recognize, such as a request for sending back a message read notification. The message priority and Reply-To address are conveniently exposed as properties of the `MailMessage` class:

```
$message.Priority = [System.Net.Mail.MailPriority] "High"  
$message.ReplyTo = test2@deshev.com
```

To accommodate setting those properties in our script, we need to add two more parameters: `$priority` and `$replyTo`. They default to `$null` and an empty string respectively, and we need to add some checks to skip configuring them if the parameters have not been provided. Here is how to do it:

```
if ($priority -ne $null)  
{  
    $message.Priority = [System.Net.Mail.MailPriority] $priority  
}  
  
if ($replyTo -ne "")  
{  
    $message.ReplyTo = $replyTo  
}
```

Figure 18-3 displays the result: a high-priority message that needs action right away!

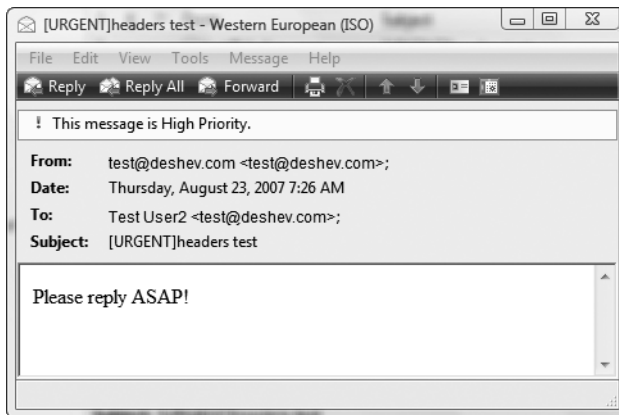


Figure 18-3. Windows Mail displaying a high-priority message

Requesting a message read receipt is not supported with a property, so we need to add a custom header for that. The header name is `Disposition-Notification-To`, and the header text should contain the e-mail address that should be notified, usually the sender address. To make our script generic enough, we add a solution that will set any header we like. We pull all header-related operations in a `Set-Headers` function that knows how to set the message priority, the Reply-To address, and any additional headers. The extra headers are passed in the `$extraHeaders`

parameter as a collection of objects with `HeaderName` and `HeaderText` properties. Again, we use `foreach` to set all headers. Here is the final `Set-Headers` function:

```
function Set-Headers($message, $priority, $replyTo, $extraHeaders)
{
    if ($priority -ne $null)
    {
        $message.Priority = [System.Net.Mail.MailPriority] $priority
    }

    if ($replyTo -ne "")
    {
        $message.ReplyTo = $replyTo
    }

    $extraHeaders | foreach {
        $message.Headers.Add($_.HeaderName, $_.HeaderText) }
}
```

We place the `Set-Headers` function in an updated version of our `mailutils` library script, `mailutils-headers.ps1`.

We have to update our driver and save it under a new name, `send-mail-headers.ps1`. It now has to declare the new `$priority`, `$replyTo`, and `$extraHeaders` parameters and pass them to `Set-Headers`:

```
param($to = $(throw "'to' cannot be null"),
      $cc = @(),
      $bcc = @(),
      [string] $subject = "(no subject)",
      [string] $body = "",
      $priority = $null,
      [string] $replyTo = "",
      $extraHeaders = @())

. .\mailutils-headers.ps1

$message = Create-Message $to $cc $bcc $subject $body
Set-Headers $message $priority $replyTo $extraHeaders
Send-Message $message
```

Now, we can send a message that requests a read receipt from its recipients:

```
.\send-mail-headers.ps1 -to "Test User2 <test@deshev.com>" `
    -subject 'important event'-body 'Event info...' -extraHeaders `
    @(@{HeaderName="Disposition-Notification-To"; `
        HeaderText="test@deshev.com"})
```

When the recipient opens the message, he or she will be asked to send back a notification response. Figure 18-4 presents the Windows Mail prompt for sending a receipt.

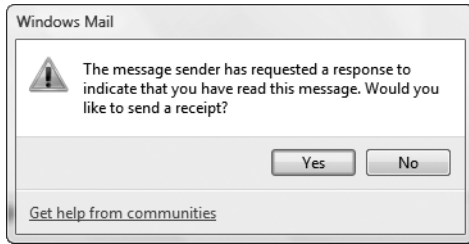


Figure 18-4. *The Windows Mail prompt for sending a read receipt*

We should not rely on receipts too much, because recipients can always choose not to notify us or just configure their e-mail clients to silently ignore read receipts.

The `$extraHeaders` parameter now allows us to pass any header we wish and tailor the message according to our environment. If users are accustomed to filtering messages according to a some header, say `Custom-Header`, we can send properly formatted messages that get processed by the correct filter with a command like this:

```
.\send-mail-headers.ps1 -to "Test User2 <test@deshev.com>" `
    -subject 'important event'-body 'Event info...' -extraHeaders `
    @(@{HeaderName="Disposition-Notification-To"; `
        HeaderText="test@deshev.com"},
        @{HeaderName="Custom-Header"; HeaderText="PowerShell test mail"})
```

Mail clients tend to hide most of the headers, as they do not contain information of the utmost importance. Windows Mail lists all headers in the Details tab of the Message Properties dialog. Outlook 2007 displays the message headers in the Internet Headers box that is a part of the Message Options dialog.

Multipart Messages

The world of e-mail has more than just plain text. The Multipurpose Internet Mail Extensions (MIME) standard allows us to create messages composed of cleanly separated parts. We can provide content in different formats, attach files, and embed media like images. Using the MIME standard, we can create and send rich messages that can cater to any demanding user needs.

Message Views

System administrators and software developers sometimes connect and read mail through a limited client or connection. It is always a good idea to send a message in plain text, so that it is readable even if you are using a console application or browsing through raw message dumps. On the other hand, we often need rich formatting that can highlight important portions of the message body or serve some other purpose. To balance those seemingly conflicting needs, we can use the `System.Net.Mail` abstraction of a message `AlternateView` and provide different views of the same message. Mail clients can determine which view to display to the user, and if we always provide a plain text view, we are guaranteed that the message will contain readable text even if in a raw text dump.

The most common use of alternate views is to provide a plain text alternative to a message that contains HTML. To implement that in our script, we need two body inputs for the two views: `$body` and `$htmlBody`. We will default to a plain text message if the user provides only `$body`, and we will error out if we get only `$htmlBody`, as we always want to have a text view. It is about time that we pull the message body configuration out of `Create-Message` and create a separate function, `Set-MessageBody`:

```
function Set-MessageBody($message, [string] $body, [string]$htmlBody)
{
    if ($body -ne "" -and $htmlBody -ne "")
    {
        Set-MultiPartBodies $message $body $htmlBody
    }
    elseif ($body -ne "")
    {
        $message.IsBodyHtml = $false
        $message.Body = $body
    }
    elseif ($htmlBody -ne "")
    {
        $message.IsBodyHtml = $true
        $message.Body = $htmlBody
    }
    else
    {
        throw "Cannot send a message with an empty body!"
    }
}
```

The real action happens inside `Set-MultiPartBodies`. That function creates the `AlternateView` objects. Each alternate view requires a body and content type. Mail readers will display only views of supported content types, so it is a good idea to get to know what programs your users are using. Anyway, always including a plain text view is the safest bet. Here is the code:

```
function Set-MultiPartBodies($message, $body, $htmlBody)
{
    $textView = [System.Net.Mail.AlternateView]::`
        CreateAlternateViewFromString($body, "text/plain")
    $message.AlternateViews.Add($textView)

    $htmlView = [System.Net.Mail.AlternateView]::`
        CreateAlternateViewFromString($htmlBody, "text/html")
    $message.AlternateViews.Add($htmlView)
}
```

Those two functions go to a new version of our mailutils library. We will call it mailutils-multipart.ps1, and we will update our driver script, naming it send-mail-multipart.ps1:

```
param($to = $(throw "'to' cannot be null"),
      $cc = @(),
      $bcc = @(),
      [string] $subject = "(no subject)",
      [string] $body = "",
      [string] $htmlBody = "")

. .\mailutils-multipart.ps1

$message = Create-Message $to $cc $bcc $subject
Set-MessageBody $message $body $htmlBody
Send-Message $message
```

Now, we can send properly formatted messages like this:

```
.\send-mail-multipart.ps1 -to test@deshev.com -subject "multipart message" `
    -body "Very important!" `
    -htmlBody "<span style='color: red; '><u>very</u> important</span>"
```

Rich mail clients will display the HTML part only. Figure 18-5 shows how our message looks in Windows Mail.

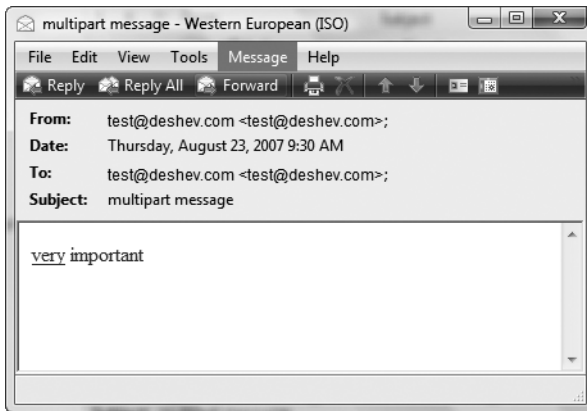


Figure 18-5. Windows Mail displaying an HTML-formatted message

Attaching Files to Messages

System automation involves dealing with lots of data, and some of that data, sooner or later, will have to be sent over e-mail. The data can vary wildly: a log file from a failed product build, a screenshot of a malfunctioning program, or an exported PDF of a sales report. Luckily, System. Net.Mail's MIME support hides all data encoding from us and makes attaching files to a

message a breeze. All we have to do is create `MailAttachment` objects and add them to the message's `Attachments` collection.

Again, we are following the philosophy of adding the file attachment feature to our script in a manner that is least disruptive to the code written so far. We will add a new optional parameter, `$attachments`, that will contain a collection of file paths. The default is an empty collection. We put all attachment-processing logic in a separate function, `Set-Attachments`, that will loop over the file paths and add them to the message.

```
function Set-Attachments($message, $attachedFiles)
{
    foreach ($file in $attachedFiles)
    {
        $attachment = New-Object System.Net.Mail.Attachment -argumentList $file
        $message.Attachments.Add($attachment)
    }
}
```

We add that function to the `mailutils` library and name this version `mailutils-attachment.ps1`. Our updated driver script, `send-mail-attachment.ps1`, needs a one-line change—we need to call `Set-Attachments` before sending the message:

```
param($to = $(throw "'to' cannot be null"),
      $cc = @(),
      $bcc = @(),
      [string] $subject = "(no subject)",
      [string] $body = "",
      [string] $htmlBody = "",
      $attachments = @())

. .\mailutils-attachment.ps1

$message = Create-Message $to $cc $bcc $subject
Set-MessageBody $message $body $htmlBody
Set-Attachments $message $attachments
Send-Message $message
```

That is all! We call the script by passing a new array of paths:

```
.\send-mail-attachment.ps1 -to "test@deshev.com" -subject "Program error" `
    -body "Program error log and details" `
    -attachments @"(C:\PowerShell\kill.log", "C:\PowerShell\test.txt")"
```

Figure 18-6 shows a message with attachments generated by our script

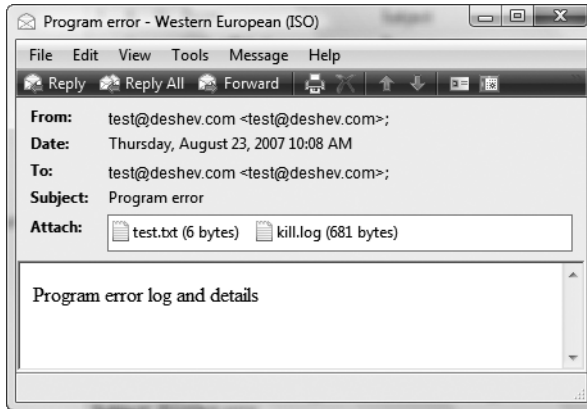


Figure 18-6. *Windows Mail showing our message attachments*

Embedding Media in Messages

HTML allows us to embed various resources in our document, with the most commonly used ones being images. A normal web page would include code like `` to make the browser load the image and display it inside the document. We can not use this approach in HTML code included in an e-mail message. Theoretically, we can request an image from a remote server, but in practice, most mail readers block such requests because of privacy concerns—most people do not want to initiate random connections to servers on the Web when they read a message. We can embed the image in the message, MIME-encode it similar to an attachment, and give it a content ID. Having a content ID, such as `image1`, will allow us to reference that image with a URL of the type `cid:image1`.

We embed files in messages by wrapping them in `LinkedResource` objects and adding those to the mail message. To create a `LinkedResource` object, we need to provide a file path and a `ContentId` setting. Note that using `LinkedResource` objects requires that we set up alternate views of our message, so we will have to pass both the `$body` and `$htmlBody` parameters. We can reference linked resources only from the HTML view.

We will add this feature to our script by creating a new function, `Set-LinkedResources`. It needs a collection of objects in which each has a `File` and `ContentId` property. The script takes a new optional parameter, `$linkedResources`, that defaults to an empty collection. Our function code follows; we first confirm that we have an HTML alternate view and provide a descriptive error message if we do not:

```
function Set-LinkedResources($message, $linkedResources)
{
    $htmlView = $message.AlternateViews | where {$_.ContentType -like 'text/html*'}
    if ($htmlView -eq $null)
    {
        throw "You need an HTML alternate view to use linked resources!"
    }
}
```

```

foreach ($resourceInfo in $linkedResources)
{
    $resource = New-Object System.Net.Mail.LinkedResource `
        -argumentList $resourceInfo.File
    $resource.ContentId = $resourceInfo.ContentId

    $htmlView.LinkedResources.Add($resource)
}
}

```

The Set-LinkedResources function goes to a new version of the mailutils library: mailutils-linkedresources.ps1. For the new driver script, send-mail-linkedresources.ps1, we just need to add a call to Set-LinkedResources and the \$linkedResources parameter declaration:

```

param($to = $(throw "'to' cannot be null"),
    $cc = @(),
    $bcc = @(),
    [string] $subject = "(no subject)",
    [string] $body = "",
    [string] $htmlBody = "",
    $attachments = @(),
    $linkedResources = @())

. .\mailutils-linkedresources.ps1

$message = Create-Message $to $cc $bcc $subject
Set-MessageBody $message $body $htmlBody
Set-Attachments $message $attachments
Set-LinkedResources $message $linkedResources
Send-Message $message

```

We call the script by providing a collection of linked resource information objects that are really plain associative arrays:

```

.\send-mail-linkedresources.ps1 -to test@deshev.com -subject 'inline image' `
    -body 'test' -htmlBody `
    '<p>Here is a sample chart:</p>' `
    -linkedResources (@{File="C:\PowerShell\chart.png"; ContentId="chart"})

```

Note how we use the snippet in our HTML code to refer to the chart image. Figure 18-7 shows how our embedded chart looks in Windows Mail.

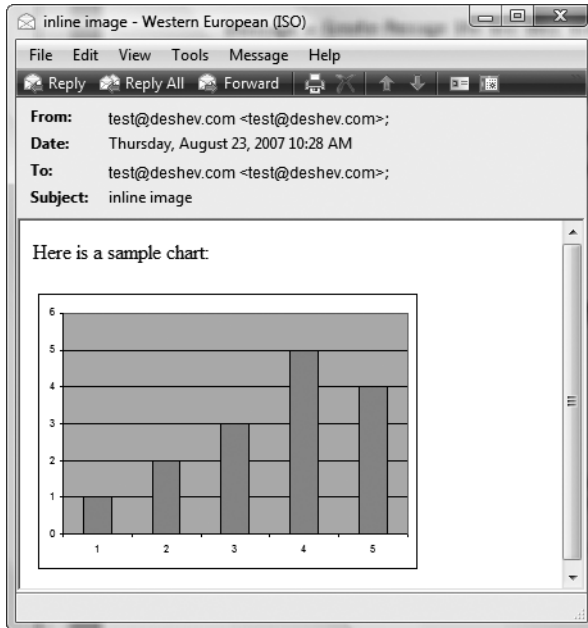


Figure 18-7. Windows Mail displaying an HTML-formatted message with an embedded image

Putting It All Together

We have gone through quite a lot! Our simple e-mail script grew into something quite sophisticated. The final script can now do the following:

- Send both plain text and HTML e-mail.
- Handle multiple recipients.
- Support carbon copying and blind carbon copying.
- Set message priority and Reply-To address.
- Add custom headers and support message read receipts.
- Attach files to the message.
- Embed media files in HTML-formatted messages.

The beauty of it all is that our script library, `mailutils-final.ps1`, contains less than 100 lines of script. The driver script, `send-mail-final.ps1`, is about 20 lines long and just calls functions from the script library. We got every feature from the preceding list for a mere 120 lines of code—talk about PowerShell's language power and expressiveness!

All features are optional, and the only mandatory parameters are `$to` and `$body`. If we try to push our script's limits and use everything, the command might look similar to this:

```
.\send-mail-final.ps1 -subject 'Everything and the kitchen sink' `
  -to "Test User <test@deshev.com>" `
  -cc "Hristo Deshev <hristo@deshev.com>" `
  -bcc "Test User2 <test2@deshev.com>" `
  -body "Here is a sample chart" `
  -htmlBody "<p>Here is a sample chart:</p><img src='cid:chart' />" `
  -attachments @"(\"C:\PowerShell\kill.log", \"C:\PowerShell\test.txt")" `
  -linkedResources (@{File=\"C:\PowerShell\chart.png\"; ContentId=\"chart\"}) `
  -priority "High" `
  -replyTo "test2@deshev.com" `
  -extraHeaders (@{@HeaderName=\"Disposition-Notification-To\"; `
    HeaderText=\"test@deshev.com\"})
```

Not impressed yet? Just look at the monster message in Figure 18-8.

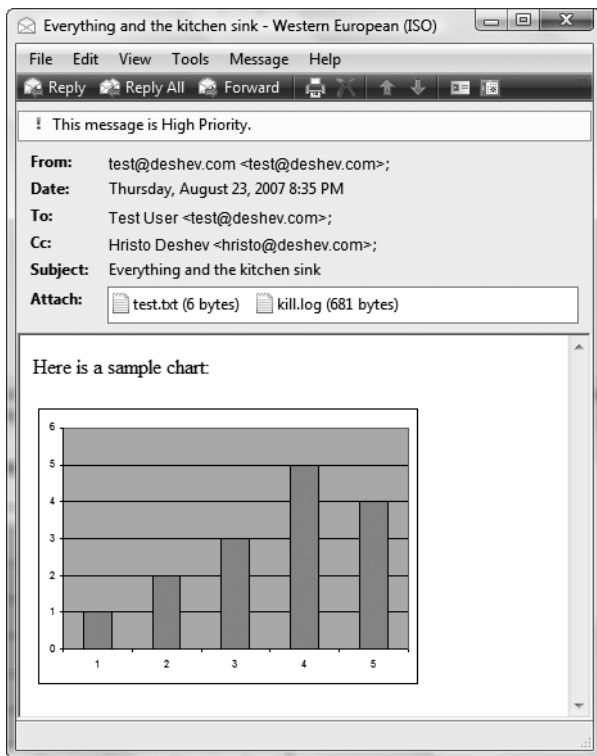


Figure 18-8. A high-priority, HTML-formatted message sent to multiple recipients, containing an embedded image, two attachments, and several custom headers

I hope neither I nor you are on the sending or receiving end of such messages! Anyway, should we want or need to, creating such a message is most certainly possible.

Summary

This chapter involved some serious programming using objects from the `System.Net.Mail` namespace in the .NET Framework. We started with simple text-based messages and went on adding support for often-used mail features like richly formatted text and attachments. Even if you are not a programmer, you can follow what the code does and get the general idea. If you do not want to get that deep, using the scripts we created does not require understanding of its inner workings. Just copy `send-mail-final.ps1` and `mailutils-final.ps1` to a convenient location, and you can use them to send e-mail on any system that has PowerShell installed. To get deeper into `System.Net.Mail` and cover advanced mail-sending scenarios, you should continue your journey with the MSDN documentation on the .NET `System.Net.Mail` classes.



Talking to COM Objects

The Component Object Model, COM for short, is a widely used platform that allows different pieces of software to interoperate without needing to know what language or programming environment has been used for implementing any of them. It has been the de facto standard for application interoperability for quite some time. The technology was invented in 1993 and started gaining popularity in 1997. Today, it is practically everywhere. COM is so popular because it makes exposing automation interfaces or extensibility points very easy. Some programs, such as Microsoft Office, use it to expose object models that can be exploited by script authors. Other programs, such as Internet Explorer and Windows Media Player, expose pieces of user interface along those object models, so those can be embedded into other programs and even web pages. COM has been extended with the DCOM system to provide distributed communication and allow objects to be deployed to remote machines. DCOM eliminates the need to know if an object runs on the same machine or not and makes it very easy to build distributed systems. For example, the WMI infrastructure that we will be discussing in the next chapter uses DCOM as its intermachine communication medium.

With the advent of the .NET Framework, Microsoft has been pushing other ways of object interoperability. The .NET Framework specifies a common type system, and that allows components to be written in different languages, and they can work with each other as if they were implemented in the same language. The .NET Remoting infrastructure provides distributed communications infrastructure that even supports more sophisticated life cycle management and object marshalling strategies than COM itself. With the release of .NET 3.0, we have yet another way for objects to communicate across process and computer boundaries—the Windows Communications Foundation. COM has traditionally been a complex system, and software developers have been reluctant to use it because of all the intricacies involved. The .NET-based component technologies are expected to replace COM, and many people have been proclaiming that COM is dead. That is really not so. Microsoft continues, and will continue, to support COM in the current and future versions of the Windows operating system. There is simply too much software out there that relies on COM for its automation and interoperability interfaces. The good news for us is that it is very easy to use COM objects from within scripting environments, and PowerShell provides excellent support for that. We do not need to deal with all the complexities involved in exposing functionality as COM objects, and we can use PowerShell to take advantage of other COM objects.

In this chapter, we will be scripting the Windows Shell and other components like the Microsoft Office tools, such as Word or Excel, and the Internet Explorer browser. I will also demonstrate how to reuse code written in other scripting languages such as VBScript or JScript, as those are based on the Windows Scripting Host technology that in turn is based on COM.

How COM Works

Everything about COM revolves around object interfaces. The component platform is intrinsically object oriented. It defines interfaces as contracts, sets of methods and properties, that objects must implement. Using contracts makes it possible for other objects to use the interfaces by knowing what operations are supported. A COM interface is a set of methods and properties that an object exposes to the outside world. It is identified by a globally unique identifier (GUID). Classes that implement those interfaces are also uniquely identified by a GUID; that GUID is also called a class ID, or a CLSID. Those classes are registered in the Windows registry, and the class IDs are the ones that the system uses when creating an object. A GUID is really a long number and is a bit hard to remember; for example, the GUID for the Internet Explorer Application object looks like this: {D30C1661-CDAF-11D0-8A3E-00C04FC9E26E}. That is why the COM creators have come up with a better way to identify an object—the program ID string, or ProgID. The ProgID is a friendly string that looks more like a class name. For example, the preceding CLSID corresponds to the `InternetExplorer.Application` ProgID.

Whenever a program wants to use a COM object, it issues a call to Windows by providing the object CLSID or ProgID. Windows locates the object and its hosting application by looking in the registry. It creates the object and asks it if it supports the interface we need right now. If it does, we are given an object reference that points to the interface implementation. We can then use that reference just like we would use a normal object. The object creation mechanism is language, process, and technology agnostic. Objects can be created in any language; they can be hosted in many runtime environments, and they do not even need to operate on the same machine. COM will know how to set up proxy objects that look just like the real object and then will marshal parameters and results back and forth, so that we do not have to care if the object really runs in our process or on a machine somewhere on the network.

How PowerShell Supports COM

PowerShell is built on the .NET Framework, and the framework already provides excellent COM support. For example, we can use the .NET infrastructure to get a hold of a .NET type that wraps a COM object and use the `Activator` class to create an instance of that type. This is how we can create an instance of the Internet Explorer browser using that approach:

```
PS> $ieType = [type]::GetTypeFromProgID("InternetExplorer.Application")
PS> $ieType.Name
__ComObject
PS> $ie = [Activator]::CreateInstance($ieType)
PS> $ie.Name
Windows Internet Explorer
PS>
```

There are two things to note in the preceding example. First, the type we get back from the `GetTypeFromProgID()` method has the odd name of `__ComObject`. All COM object wrappers are of that type. Second, we create the actual object by passing that type to the `CreateInstance()` method on the `Activator` class; this is just what we would do if we wanted to instantiate any .NET class. As you can see from the value of the `Name` property, the object we got back is an instance of the Internet Explorer browser. We will use that object to do some real-life work later in this chapter.

The preceding solution is not very good looking, is it? Two lines of code are way too many to create an object. We mentioned that we are using the `Activator` object just like we would use it for creating a .NET object. Does that mean that we can use PowerShell's standard approach for creating .NET objects, the `New-Object` cmdlet? The answer is both yes and no. The `New-Object` cmdlet can create COM objects by using their ProgID's, but it needs to be told explicitly that it has to create a COM object instead of a .NET one. To do that, we provide the ProgID as the `ComObject` parameter value. Here is how we can rework the preceding sample and make it use `New-Object`:

```
PS> $ie = New-Object -ComObject InternetExplorer.Application
PS> $ie.Name
Windows Internet Explorer
```

Now, that is way better! Besides being shorter and more readable, the code does absolutely the same thing as the previous example.

The PowerShell type system provides a built-in adapter for COM objects that knows how to query COM type information. It gets details about objects from the COM type libraries and makes all objects look as close to native .NET objects as possible. It really does a good job at that.

UNSUPPORTED: MONIKERS, RUNNING OBJECTS, AND EVENTS

Creating and accessing objects is not the entire story when working with COM. COM objects can be uniquely identified by a name or a path, similar to a URL, that is called a moniker. You can use such monikers, for example, to get a reference to an Excel worksheet or the second paragraph in a Word document. Another useful COM feature is the running objects table, a central place where objects get registered that provides us with a means to get the object. For example, the Microsoft Word application instance gets registered in that table when Word is started, thus the application object is made available to third-party code. Just like .NET objects, COM objects can expose events to their clients and raise them when some condition occurs.

All of those features are not available to PowerShell. They are relatively rarely used, and the shell designers have decided not to implement them for the first version. For now, the only level of support that we get is the `New-Object` cmdlet. Mind you, that is not bad at all—in this chapter, we will use it to accomplish lots of useful things.

Scripting Programs Through Their COM Interfaces

The primary use of COM that we will look at in this chapter is the ability to use it to automate other programs. COM has long been software developers' tool of choice to provide automation interfaces to advanced users, administrators, and other programs. A software product will typically expose its application services as COM objects that can be used from any language. The best example of that are the Microsoft Office tools; all programs in the suite expose their application objects, their documents, and other services as COM objects. We can use those objects to create and manipulate documents and all sorts of data objects.

Automating Microsoft Word

We will be using the application as the entry object when working with Microsoft Word. It is identified by the `Word.Application` ProgID. The core operations that a Microsoft Word application

exposes via its automation interface involve working with documents and text. We can open a document file, read or modify the text inside it, and then save our changes. The objects that we will be working with most of the time are Range objects. They represent a subset of the text that can be modified as a single unit; we can append or remove text, set a different font, switch to another text or background color, and so on. We can create or obtain text range objects from properties exposed by the Document object; we can get them from the currently selected text; or we can create them ourselves. No matter how we got our hands on a text range object, we can manipulate it in the same way.

In this chapter, we will be using Microsoft Word 2007. Do not worry too much if you have an older version of the program. Microsoft is famous for keeping its public objects as backwardly compatible as possible, and there is very little difference in the available methods and properties between objects exposed by Word 2007 and Word 2003, for example. The sample code will run against Word 2003 with little or no modification.

Opening Documents and Extracting Text

Let's start with the simplest Word-related task that we may have to do—reading text from a document file. A lot of tasks in a typical enterprise environment involve generating documents, sending them to other people for approval, and storing them in a central location. Being able to retrieve text from many documents is an important ability; once we can do that, we can build small scripts that can search for keywords, extract related information from many documents, build a report, merge documents, and much more.

As an example, let's have a simple document called `SampleReport.docx`, which contains two paragraphs, each having two sentences. Figure 19-1 shows how the document looks when we open it in Microsoft Word.

To get to the text in that document, we need to obtain an instance of the `Word.Application` object. It will allow us to open the document and get a reference to the document object. Once we do that, we can get the entire text using the Document's `Content` property, which will return a Range object that we can use to extract the text. We will save the code in a script file called `Get-DocumentText.ps1`. Here is the code:

```
$wordApp = New-Object -COM Word.Application
$file = (dir SampleReport.docx).FullName

$doc = $wordApp.Documents.Open($file)
$text = $doc.Content.Text
$text

$wordApp.Quit()
```

Just as you would expect, we get the text of the document from the Range object's `Text` property. Another thing to note is that we need to pass an absolute path to the document file.

Caution The Documents collection `Open` method needs a full path; a relative one will not work, as Microsoft Word will use a different folder as the current folder instead of the one you are actually in. Most likely, that will be `C:\Windows\System32`, and we definitely do not want to store our documents in there.

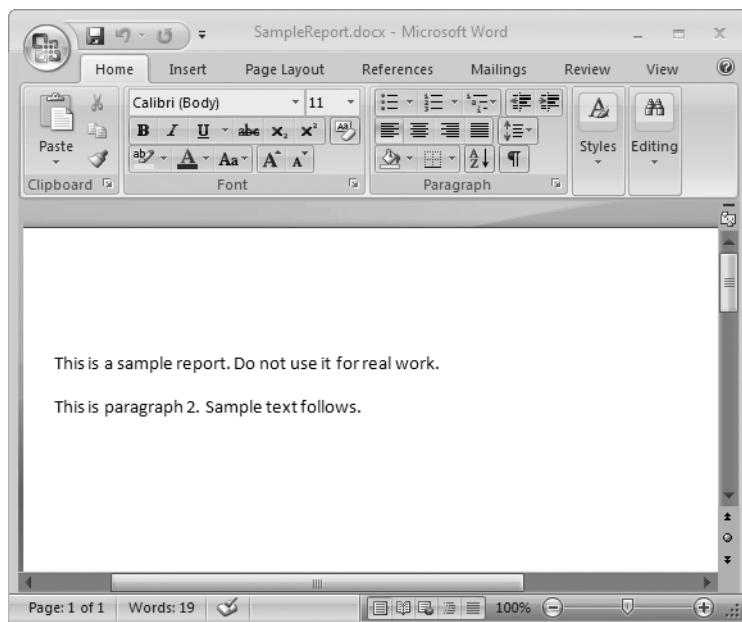


Figure 19-1. Microsoft Word showing the sample report document

Note The last line in the `Get-DocumentText.ps1` script is very important! We need to clean up and exit the application once we are done with it. Failing to do so will leave us with a hung `WINWORD.EXE` process that does nothing but take up memory. Even worse, failing to exit may keep a reference to an open document and that may lock the file, thus preventing people from using it.

Here is what happens when we run our script:

```
PS> .\Get-DocumentText.ps1
```

```
This is a sample report. Do not use it for real work.This is paragraph 2. Sample text follows.
```

We get the text all squashed inside one line; this is a side effect of transforming the rich formatted text to a plain string. If we need the text split into paragraphs, we can use the `DocumentParagraphs` collection; it will return a number of `Paragraph` objects, each representing a paragraph in the document. Each of those objects contains a `Range` that we can use to get to the actual text. Let's now create a new script, `Get-DocumentParagraphs.ps1`, that gets all paragraphs in our document. Here is the code:

```
$wordApp = New-Object -COM Word.Application
$file = (dir SampleReport.docx).FullName

$doc = $wordApp.Documents.Open($file)
```

```

$paragraphs = [object[]] $doc.Paragraphs
foreach ($paragraph in $paragraphs)
{
    Write-Host "PARAGRAPH:"
    Write-Host $paragraph.Range.Text
}

$wordApp.Quit()

```

The preceding code is pretty straightforward. There is only one minor quirk—the `[object[]]` cast that converts the `Paragraphs` collection object to a PowerShell collection. We need that so that we can iterate the collection using the `foreach` statement or access items inside by index. Here is what we get after running the script:

```

PS> .\Get-DocumentsParagraphs.ps1
PARAGRAPH:
This is a sample report. Do not use it for real work.
PARAGRAPH:
This is paragraph 2. Sample text follows.

```

We can navigate the document in smaller chunks; the `Document` object exposes a `Sentences` collection that allows us to get ranges pointing to sentence objects recognized by the application. We can use that to create another script, `Get-DocumentsSentences.ps1`, which will get all sentences' text. Here is the code:

```

$wordApp = New-Object -COM Word.Application
$file = (dir SampleReport.docx).FullName

$doc = $wordApp.Documents.Open($file)

$sentences = [object[]] $doc.Sentences
foreach ($sentence in $sentences)
{
    Write-Host "Sentence:"
    Write-Host $sentence.Text
}

$wordApp.Quit()

```

The code looks pretty similar to the example that iterates over paragraphs. The only difference is that the `Sentences` collection contains raw `Range` objects. Unlike paragraphs, Word does not use a separate object to represent sentences—do not start looking for that `Sentence` object. Here is what we get once we run the script:

```

PS> .\Get-DocumentsSentences.ps1
Sentence:
This is a sample report.
Sentence:
Do not use it for real work.
Sentence:

```


This is paragraph 2.

Sentence:

Sample text follows.

The smallest unit of text that we can get is the word. We can do that from the `Words` collection. We can get all words in another script, `Get-DocumentWords.ps1`:

```
$wordApp = New-Object -COM Word.Application
$file = (dir SampleReport.docx).FullName

$doc = $wordApp.Documents.Open($file)

$words = [object[]] $doc.Words
$words | select Text

$wordApp.Quit()
```

Just like with sentences, the `Words` collection returns a collection of `Range` objects. The preceding script returns a collection containing the ranges' `Text` property values. Here is the output that we get after running it:

```
PS> .\Get-DocumentWords.ps1
Text
----
This
is
a
sample
report
.
Do
not
...
```

Creating and Modifying Documents

The text range objects that we have been using so far are very powerful. We have been using them to get a read-only view of the document, but now we are going to change that. Ranges can have their text modified; their text can be reformatted independently; and much more. And we can use them to add content to our document. The simplest useful scenario that comes to mind is saving a snippet of text to a Word document. You may already be guessing at the approach: get the `Document.Content` range, and set its `Text` property. That is basically the idea. The only additional tasks are to create a new document and save it as a file once we are done modifying the text. We will create a new script, `New-Document.ps1`, that will accept two parameters: the path to the new document and the text that will be saved inside. Here is the code:

```
param ($path, $text)

$wordApp = New-Object -COM Word.Application
```

```
#delete the file if it already exists
if (Test-Path $path)
{
    del $path
}

$doc = $wordApp.Documents.Add()
$doc.Content.Text = $text
$doc.SaveAs([ref] $path)

$wordApp.Quit()
```

The code calls the `Documents.Add()` method to tell Word to create a new document object. Once it sets the text to the `Content` range, it saves the document to disk by calling the `SaveAs()` method. The `SaveAs()` method has one small peculiarity we need to take into account—it needs to have the string variable that holds the path to the file as a reference. That is why we need the `[ref]` cast. Here is how we call the script:

```
PS> $newDoc = Join-Path (Get-Location) Test.docx
PS> .\New-Document.ps1 $newDoc "Hello Microsoft Word!"
PS>
```

We get no text output back, and we need to open the `Test.docx` file to see the final result. Figure 19-2 shows how the file looks.

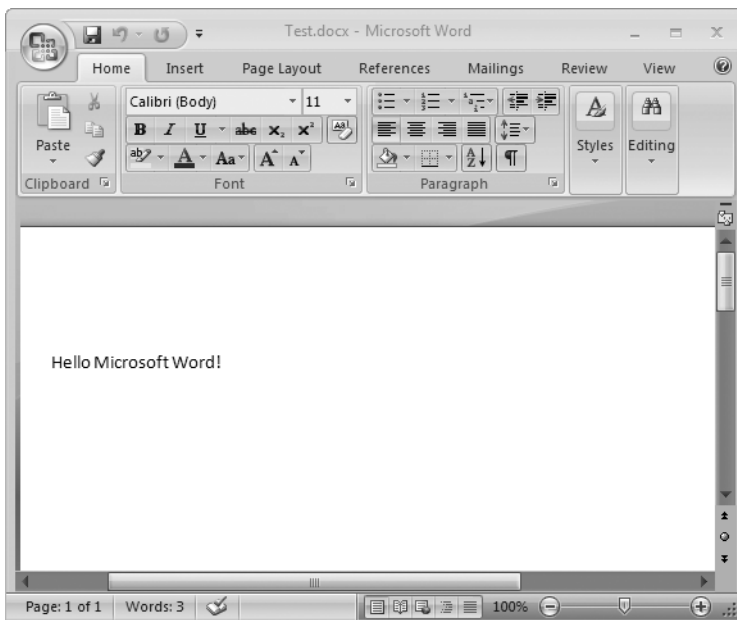


Figure 19-2. Microsoft Word showing the `Test.docx` generated document

Apart from creating new documents, we can modify existing ones by opening them first. All we need to do is use the code we saw in the text retrieval examples. After we open the document, we can locate the part we need (a Range object, of course), modify it, and then save the document to disk. To show that we can modify not only the text content but formatting too, we will create a script that opens a document, appends some text at the beginning, and colors it in red. We will call the script `Insert-RedText.ps1`. It will modify a copy of our original `SampleReport.docx` file that we will name `SampleReportRed.docx`. Here is the code:

```
$wordApp = New-Object -COM Word.Application
$file = (dir SampleReportRed.docx).FullName

$doc = $wordApp.Documents.Open($file)

$documentFront = $doc.Content
$documentFront.End = $start.Start

$documentFront.Text = "IMPORTANT!!! Please review!`n"
$documentFront.Font.Name = "Comic Sans MS"
$documentFront.Font.ColorIndex = 6

$doc.Save()
$wordApp.Quit()
```

The insertion technique needs some explaining. Initially, we get a range that represents the entire document contents. We then collapse the range by setting its `End` property to the value that the `Start` one contains. This effectively creates a zero-length range that is positioned at the beginning of the document. We can use it as an insertion point; setting its `Text` property will insert text at the top of the document. We add the “IMPORTANT!!! Please review!” text followed by a new line symbol. We then use the range object to change the text font to what is probably the most annoying font in the universe: Comic Sans MS. We also change the text color to dark red; that is what the 6 color index stands for. Finally, we call the `Save()` method, so that our changes are persisted. Running our script does not generate much text output:

```
PS> .\Insert-RedText.ps1
PS>
```

To see the result of our work, we must open the `SampleReportRed.docx` file. Figure 19-3 shows the document as displayed by Microsoft Word.

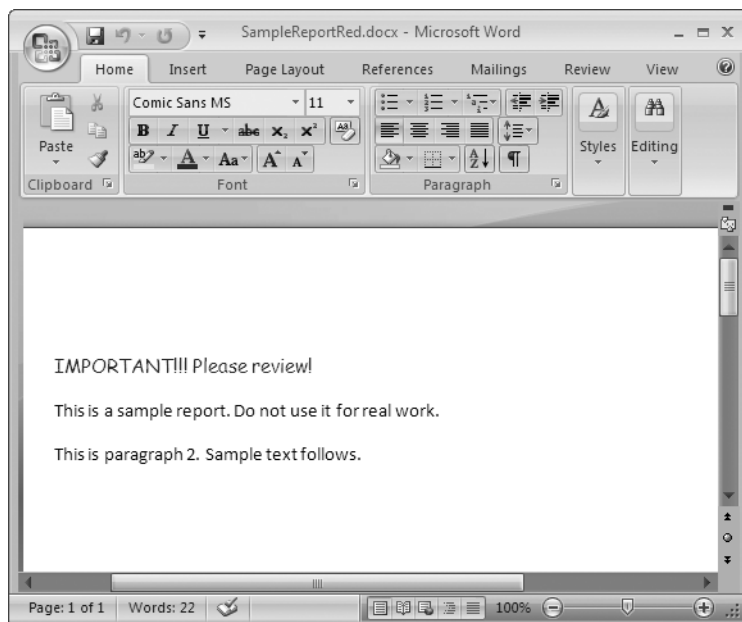


Figure 19-3. *Microsoft Word showing the text we inserted at the beginning of the document*

DISPLAYING THE WORD WINDOW FOR DEBUGGING PURPOSES

Automating Microsoft Word from a script does not display the application window—that is the entire point. The user should not see any windows, and the task at hand should be performed in the background. While that is the desired behavior at the end of the day, being unable to view the application window may sometimes get in the way. Imagine debugging a tough problem in your code and having to switch to a folder window and open the file your script just modified just to find out if the script worked correctly. To deal with that problem more efficiently, you can instruct Word to display its application window. This way, you get the best of both worlds—the script code continues to run and modify the document, and you see the end results.

To force the main application window to become visible at runtime, you need to just set the `Application` object's `Visible` property to `$true`. Using the variable name from the scripts in this section, you can insert the following code anywhere in your script:

```
$wordApp.Visible = $true
```

Make sure you get rid of that code once you are done and your script works perfectly!

Using Microsoft Word's Spell Checker

Word is really a wonderful program. It contains features, such as a powerful spell-checker, that people cannot live without. The greatest part about the spell-checker is that it is available through the COM automation interface. We do not have to do anything special to use it—all we have to

do is add some text and get the Document SpellingErrors property value. The property will contain a collection of Range objects that contain the erroneous text portions. Once we have a range that contains an incorrect word, we can get helpful suggestions for correction by calling the Range object GetSpellingSuggestions() method. That method will return a collection of SpellingSuggestion objects whose Name property values contain the text of the suggested correction.

Armed with the theory I outlined previously, we can create a script that passes a chunk of text to Microsoft Word and reports on any spelling errors that it finds. We will call the script SpellCheck-Text.ps1. Here is its code:

```
$wordApp = New-Object -COM Word.Application
$doc = $wordApp.Documents.Add()
$doc.Content.Text = "this is some mistakn text"

$errors = [object[]] $doc.SpellingErrors

foreach ($errorRange in $errors)
{
    Write-Host "Error: $($errorRange.Text)"
    $suggestions = ([object[]]$errorRange.GetSpellingSuggestions())
    $suggestedWords = $suggestions | foreach { $_.Name }
    Write-Host "Suggestions :"
    Write-Host $suggestedWords
}

$wordApp.Quit()
```

Again, we use a cast for the COM collection that the SpellingErrors property returns to a PowerShell object array, so that we can iterate over it with a foreach loop. We get the SpellingSuggestion objects for each error range object and turn them to an array using the same cast technique. We then pipe the collection through the foreach cmdlet, which will return a collection that contains only the Name property values. Here is the result we get after running the script:

```
PS> .\SpellCheck-Text.ps1
Error: mistakn
Suggestions :
mistaken mistake mistaking
PS>
```

Getting Help and Going Further

The examples we went through barely scratched the surface of automating Microsoft Word. Entire books have been written on this topic. Experienced scripters or people with background in Microsoft Office automation will recognize the objects we have been using in our scripts, because those are the very same objects that we manipulate from the Visual Basic for Applications (VBA) environment that is built in all Microsoft Office applications. The VBA programming

language is different than PowerShell, but it is not *that* different. Most of the time, translating a code snippet from VBA to PowerShell is pretty straightforward, as all the method invocations and property access syntax is similar. The only thing that can bite you is accessing collections, and you already know how to work around that by using a cast to convert them to PowerShell collections. Armed with that knowledge, you can use the “Microsoft Word Visual Basic Reference” section in Microsoft Word’s help to get information about automation object and to find useful code snippets.

Another strong ally in Microsoft Office automation is making Word generate your code. Whenever you want to automate a task, just use the macro recording facility. Record your actions, and then open the macro in the Visual Basic code editor. Microsoft Word will have generated the Visual Basic code that executes the actions you performed while recording. Translating that code to PowerShell should be relatively easy.

Scripting Microsoft Excel

Just like with Microsoft Word, the Excel automation entry object is the application one. It has the ProgID of `Excel.Application`. Once we create it, we can use it to get to a `Workbook` object using the `document.Workbooks` collection. `Workbooks` have a collection of `Worksheet` objects that they make available through the `Worksheets` property. Once we get hold of a worksheet, we can get a cell or a range of cells.

As with text ranges in Microsoft Word, cell ranges are a way to refer to a group of cells, so that we can modify their formatting and other properties in common. A `Range` object may contain one or many cells, while a range contains a collection of cells that we can access programmatically.

Reading Cell Values from a Document

If I had to pick only one operation that we could do when automating Microsoft Excel, it would probably be the ability to retrieve cell values from within a document. We can do that using the `Cells` property on the `Worksheet` object. It contains a COM collection that we can use to get a cell by its row and column index. COM collections are a bit odd; by convention, they are indexed through the `Item` method. Languages like Visual Basic handle that method gracefully, and the collection can be indexed in a manner similar to the bracket notation we use for indexing PowerShell collections—the only difference being that Visual Basic uses parentheses instead. Another thing to keep in mind is that COM collections follow the time-honored Visual Basic traditions when it comes to indexing items. The first item is available at index 1, whereas the standard .NET and PowerShell collections use the index 0 for that.

As an example for reading cell values from a workbook, we can imagine that we have a list of FBI agents. The list is saved in a workbook called `Agents.xlsx`, and it uses two columns: one that contains agent first names and another, the last names. Figure 19-4 shows how that document looks in Excel.

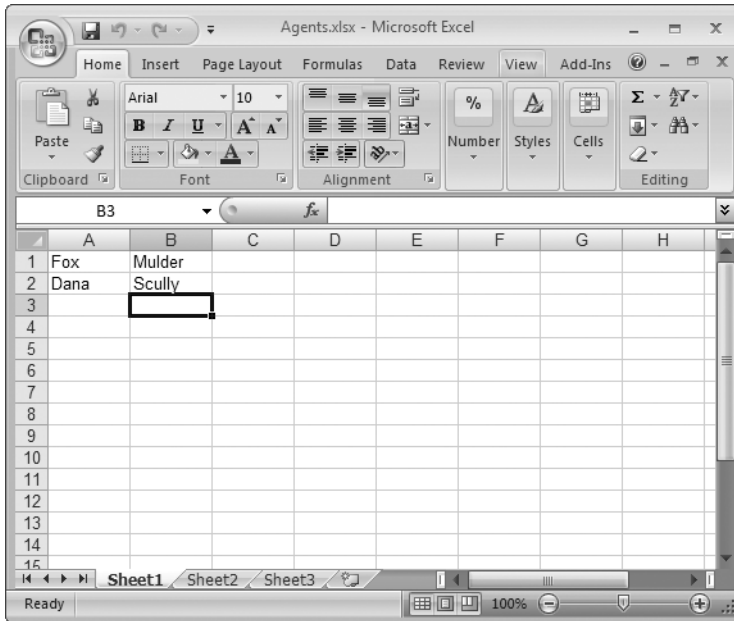


Figure 19-4. Microsoft Excel showing our agent list

We will be creating a script called `Get-NonEmptyCells.ps1` that will read the agents' names from the document. Since we do not know how many records we have, we will loop over the worksheet rows until we get a line that has an empty value for either the first or the last name. Let's take a look at the code:

```
$excelApp = New-Object -COM Excel.Application
$file = (dir Agents.xlsx).FullName

$book = $excelApp.Workbooks.Open($file)

$sheet = $book.Worksheets.Item(1)

$row = 1
while($true)
{
    $firstName = $sheet.Cells.Item($row, 1).Value2
    $lastName = $sheet.Cells.Item($row, 2).Value2

    if (!$firstName -or !$lastName)
    {
        break;
    }
}
```

```

        "$firstName $lastName"
        $row++
    }

$excelApp.Quit()
$book = $null
$sheet = $null
$excelApp = $null
[GC]::Collect()

```

Just as with Microsoft Word documents, we again need to provide a full path to our workbook. The code opens the workbook and gets the first worksheet. Note that it gets the worksheet at index 1, as the collection is one based. After that, we need a loop that will go through the rows one at a time. We get the cell objects through the worksheet `Cells` property; we use the `Item` method to get the cell at the specified row and column index. The easiest way to get a cell object's text is to use its `Value2` property. A cell can contain all types of values, and if we wanted to get those, we would use the `Value` property and convert that to the type of object we are after. Since we are working with text only, we will use `Value2` as a simpler way to get a cell's text content.

An important thing to note here is that we need to clean up after we are done. We start by instructing the application to exit by calling its `Quit()` method. Unfortunately, we need to do much more—we need to explicitly set to `$null` all variables that may point to an Excel COM object. We also need to trigger garbage collection explicitly by calling `[GC]::Collect()`; doing so will finally make the Excel process exit.

Note Cleaning up after a script that uses Excel may not be a walk in the park. Due to the nature of memory management in COM, an object will stay in memory until there are no clients holding references to it. The .NET object wrappers that PowerShell uses hold a reference to objects, and that may prevent the Excel process from exiting, even when we call the application `Quit()` method. This is a limitation of Excel automation and has been plaguing script authors for several years. For example, there is a Microsoft Knowledge Base article (KB317109) documenting the issue that dates back to September 2004.

To make sure that Excel exits cleanly after your script finishes, always set all your variables that point to Excel objects to `$null`. There is no good way to find a variable that you forgot to set to null once you have a hung Excel process. The best way to deal with that is to extract data from the COM objects and set the variables that point to them to `$null` as soon as you are done with them. Even if you do that, do not forget the `[GC]::Collect()` call at the end of your script. It will release the object wrappers, and if you have done a good job at setting variables to `$null`, Excel will exit.

Finally, here is the output that we get from our script—a string containing the agent's full name for every line in the worksheet:

```

PS> .\Get-NonEmptyCells.ps1
Fox Mulder
Dana Scully

```

Caution Excel is extremely sensitive to the Windows regional settings, and a lot of features depend on information extracted from those settings: number and date formats, sorting rules, and so on. Excel relies on components, called multilingual user interface packs, to provide support for different cultures' specific settings. Automating Excel from a system having a specific culture configured in its regional options control panel requires that the corresponding multilingual user interface pack be installed on the machine. If it's not installed, you may get a cryptic "Old format or invalid type library." error message, and your script will not run. This often happens when Excel has been installed with its English (United States) interface pack, and then the system has been configured with another default language, say English (United Kingdom). To fix this problem, you have to either install the English (United Kingdom) multilingual user interface pack or switch the system back to English (United States) from the control panel. The problem is documented in detail in a Microsoft Knowledge Base article available at <http://support.microsoft.com/default.aspx?scid=kb;en-us;320369>.

Very often, we know the exact location of our data, and we do not need to loop over all cells until we get the values we need. In the case of our agents list, we know that the data we need is in the A1:B2 range. We can pass that to the worksheet `Range()` method and get a `Range` object back. We can then loop over the cells in the range and get their values. To demonstrate that, let's create another script, `Get-CellsInRange.ps1`. Here is its code:

```
$excelApp = New-Object -COM Excel.Application
$file = (dir Agents.xlsx).FullName

$book = $excelApp.Workbooks.Open($file)

$sheet = $book.Worksheets.Item(1)
$cellRange = $sheet.Range("A1:B2")

$cells = [object[]] $cellRange.Cells
$cells | select Value2

$excelApp.Quit()

$cellRange = $null
$cells = $null
$book = $null
$sheet = $null
$excelApp = $null
[GC]::Collect()
```

We pass the range of cells that interests us to the worksheet `Range()` method. We get back a range object that contains the cells in question. We use the standard `[object[]]` cast trick to convert the result to a PowerShell collection. We pass that collection through the `select` cmdlet, which, in turn, gets just the `Value2` property value. The script ends with the now familiar procedure of exiting the application, setting all variables to `$null`, and triggering garbage collection.

Except for the clumsy clean up, our script works just fine. Here is what we get back when we run it:

```
PS> .\Get-CellsInRange.ps1
```

```
Value2
-----
Fox
Mulder
Dana
Scully
```

Modifying Workbook Content

Most of the time, we do not need Excel automation if we just want to import some data in a workbook; PowerShell can export any object to a comma-separated file with its `Export-Csv` cmdlet. There are cases, though, where we need to modify an existing document or dabble with Excel-specific functionality like changing a cell's appearance. This is where we have to reach for COM automation in our virtual bag of tricks.

To demonstrate how we can fill a workbook with some data, we will be creating a script, called `Export-ProcessesToExcel.ps1`, which will fill the first two columns with the process ID and process name, respectively. We can set the actual cell text through the `Value2` property. At the end, to show off and make the text stand out, we will select the first two columns and bold the text and color it in red. Without making you wait any longer, here is the script code:

```
$excelApp = New-Object -COM Excel.Application
$file = Join-Path (Get-Location) "Processes.xlsx"

$book = $excelApp.Workbooks.Add()

$sheet = $book.Worksheets.Item(1)

$processes = Get-Process

#set cell contents
for ($i = 1; $i -le $processes.Count; $i++)
{
    $processIdCell = $sheet.Cells.Item($i, 1)
    $processIdCell.Value2 = $processes[$i].Id
    $processNameCell = $sheet.Cells.Item($i, 2)
    $processNameCell.Value2 = $processes[$i].Name
}

$processIdCell = $null
$processNameCell = $null

#bold the A and B columns and make their text red
$columnsABRange = $sheet.Range("A:B")
$columnsABRange.Font.Bold = $true
$columnsABRange.Font.ColorIndex = 3
$columnsABRange = $null
```

```

$book.SaveAs($file)
$excelApp.Quit()

$book = $null
$sheet = $null
$excelApp = $null
[GC]::Collect()

```

We get process information using the `Get-Process` cmdlet (refer to Chapter 14 for more on working with it). The really interesting part of the script is the moment where we create a range that contains the first two columns. We use the Excel `A:B` cell range syntax to do that. Modifying the font settings for the range object will affect all the cells it contains. Note the necessary evil of adding all those assignments to `$null` and the garbage collection trigger. I really hope that problem gets resolved in the future; our sample script would be about half its size if we did not need to manually clean our variables.

Running the script does not produce much output. We need to open the `Processes.xlsx` workbook that it generates to see if it really worked. Figure 19-5 shows how it looks.

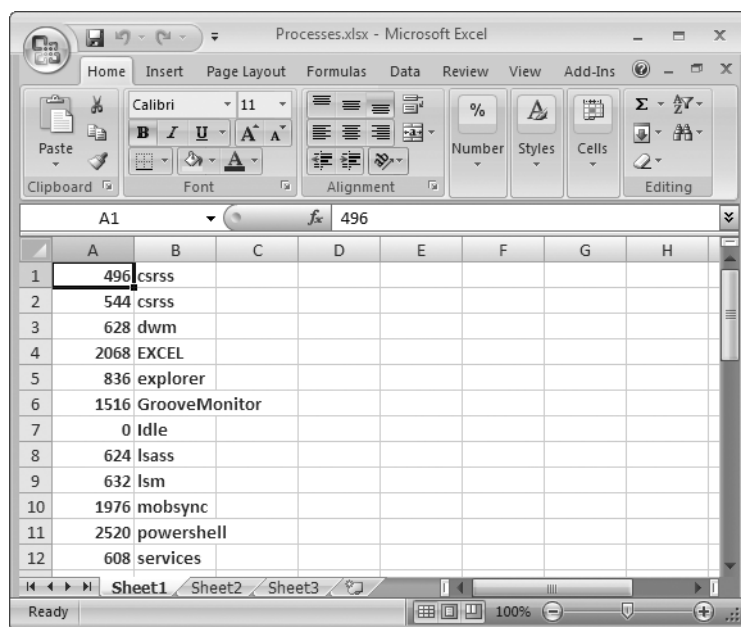


Figure 19-5. Microsoft Excel showing our list of processes

Going Further

Excel is a very complex application, and you can do many wonderful things using its automation facilities. Just as is the case with Word automation, the objects exposed to PowerShell are the same objects that are available to VBA code. Make sure you go through the “Visual Basic Reference” section in the product help.

Driving Internet Explorer

Being able to automate a web browsing session can yield a lot of benefits for many people. In Chapter 17, you saw how to perform raw HTTP requests against web servers; those techniques were appropriate for downloading a single file or a document. Often, getting information from a web site involves more complex actions. At the simplest, many sites require authentication that involves filling out a form and submitting it to the server. We can push our knowledge of the HTTP protocol to the limit and do that by crafting a proper HTTP POST request that contains all the form values. We could then parse the response, get the session cookie value, and use that for all subsequent requests. While that can be done, it is tedious, hard, and error prone. Luckily, the Internet Explorer browser provides a rich object model based on COM that we can use to initiate and script a browser session. We can do almost anything we would have done in person through the automation interface. This is so convenient that people have built web site test tools that automate Internet Explorer to simulate user interaction: typing text, clicking buttons, and much more. We can use the browser to build scripts that submit information to a server or extract values from a web page.

A script entry point for scripting Internet Explorer is again the application object, and its ProgID is `InternetExplorer.Application`. The automation paradigm we need to follow goes like this:

1. Create an Internet Explorer application object.
2. Navigate to a URL.
3. Wait for the browser to load the document contents.
4. Perform any actions on the document.
5. Exit the application by calling the `Quit()` method.

Of all the steps, the third one needs the most explaining. Internet Explorer navigates and downloads web page content in the background. That means that our script will not get blocked, and it may crash if it tries to access the document contents before the document is loaded—that is why we have to wait. We do that by blocking execution using the `sleep cmdlet` and periodically checking if the application `Busy` property has turned to `$false`. Only then it is safe to access and modify the web page document.

Scripting a Browser Session

Let's now use the Internet Explorer automation objects to open a browser window and perform a search with Microsoft's Live search engine available at <http://www.live.com>. We will create a script, called `Search-LiveCom.ps1`, which will instantiate an Internet Explorer application, navigate to the search engine page, fill in some text in the search box, and click the Search button. It will then display the browser window, let us inspect the search results, and continue browsing from there if we choose to. Before you continue and look at the script code, you may need to go through the Internet Explorer 7 protected mode sidebar, which describes the security settings under Windows Vista that may break your script.

INTERNET EXPLORER 7'S PROTECTED MODE MAY BREAK YOUR SCRIPT

Internet Explorer 7 has an enhanced security mode, called protected mode, which works only on Windows Vista. Having that enabled—and mind you, it is enabled by default—will break your scripts. The problem will manifest as the sites your script tries to navigate to open in new windows. The new windows belong to another Internet Explorer process, and you cannot manipulate those. The application object you originally created will eventually disconnect from the real browser instance, and you may get the dreaded “The RPC Server is unavailable” error. Internet Explorer’s protected mode mandates that opening a site that is in a different security zone than the one the current window is browsing should always open in a new window. That also starts a new browser process; this way, malicious code executing in one security zone does not have the slightest chance of manipulating objects running in another security zone. Unfortunately, that breaks our scripts.

We have two ways to fix that: disable protected mode or add the site to our Trusted Sites zone. The first option is the easiest. Just open the Internet Options dialog, go to the Security tab, and uncheck the Enable Protected Mode check box. Figure 19-6 shows the security tab focusing the protected mode check box.

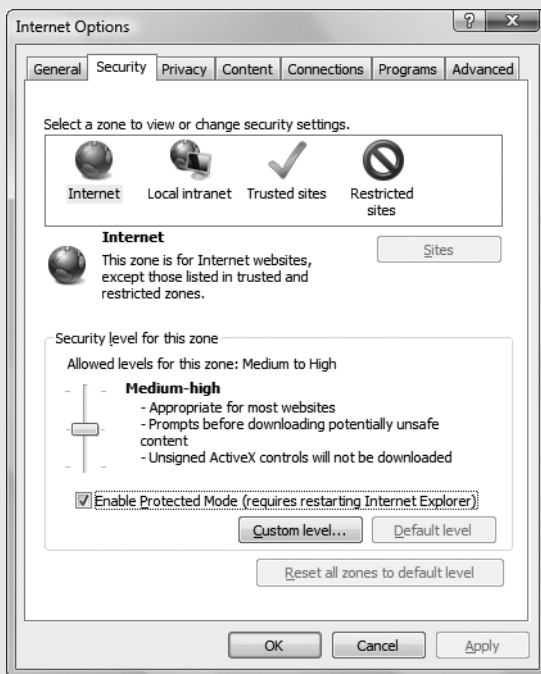


Figure 19-6. Internet Explorer's Security settings tab and the protected mode check box

Disabling protected mode may be too much for most people. After all, there must have been a real reason the Internet Explorer developers implemented that feature in the first place. If we would not want to compromise our security just for the sake of being able to run a script or two, we could add the sites that we are visiting from our scripts to the Trusted Sites zone. To do that, you must select the Trusted Sites zone in the Security tab shown in Figure 19-6 and click the Sites button. Our scripts will be automating Live.com searches, so we must add two sites to the list: <http://www.live.com> and <http://search.live.com>. Figure 19-7 shows how the configuration must look.

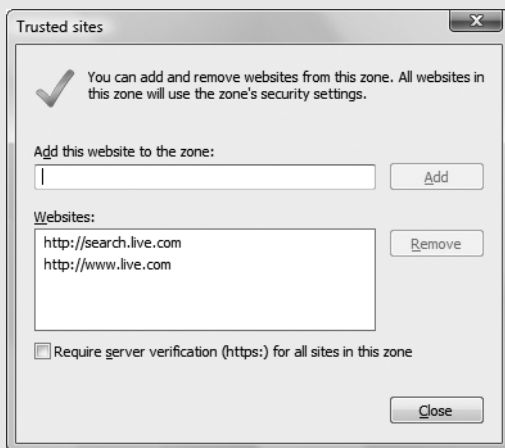


Figure 19-7. *The Live.com sites added to the Trusted Sites security zone*

Note that both sites are added with their `http://` protocol URL, and that requires that you uncheck the “Require server verification (https:)” check box.

Here is the code for our Search-LiveCom.ps1 script file:

```
function WaitForLoad ($ie)
{
    while ($ie.Busy)
    {
        sleep -milliseconds 50
    }
}

$ie = New-Object -COM InternetExplorer.Application
$ie.Navigate("http://www.live.com")
$ie.Visible = $true
WaitForLoad($ie)

$searchBox = $ie.Document.getElementById("q")
$searchBox.value = "`"Pro Windows PowerShell`""
$goButton = $ie.Document.getElementById("go")
$goButton.click();
```

There are two interesting things about the script: the `WaitForLoad` function and the text box and button interaction. The `WaitForLoad` function checks if the browser is busy every 50 milliseconds. If it still is, it sleeps for another 50 milliseconds. We need the `sleep` call so that we do not hog the entire CPU resources by doing a busy loop. Our users will not notice a delay of up to 50 milliseconds after their page finishes loading while the script is sleeping.

Once the page loads completely, we can start working with objects inside the document. We get a real HTML document object from the Document property; it supports all the methods and properties that a web developer would have available had he or she been writing JavaScript code for the web page we just loaded. We use the `getElementById()` method to get a reference to the textbox with the `q` ID—it is the one that holds the search query text. The Search button has an ID of “go”, and we use it to get a reference to that object just like we did for the text box. We set the text box text via its `value` property; we will be searching for the “Pro Windows PowerShell” string, surrounded with quotes to get an exact phrase match. To start the actual search, we call the `click()` method on our button that submits the form data to the server. Initially, the Internet Explorer window is not visible, but we set its `Visible` property to `$true`, so that we can get a look at our search results. Figure 19-8 shows the end result.



Figure 19-8. Internet Explorer showing our Live.com search results

Extracting Data from the DOM Tree

Quite often, we do not want to display the browser window at all. All we want is a completely unattended web session that will fetch some data from a remote server and not bother the user in any way. To extract data from the document, we need to use the HTML document object model (DOM) programming interface. It is a set of standard interfaces defined by the W3C that should be available to JavaScript code running in any browser. Readers with some web scripting background might have noticed the familiar `getElementById()` method calls in the previous example. We can use the entire HTML DOM programming model that Internet Explorer implements from our scripts. We can do that because all DOM objects in Internet Explorer are implemented as COM objects, and that opens them up to the entire Windows scripting world.

Tip The DOM programming interfaces are documented both by the W3C and Microsoft. The W3C specifies how things should work for all browsers. In reality, no browser implements the standard to the letter, and using Microsoft's documentation is best, as it covers Internet Explorer's implementation. Since Internet Explorer is the only browser scriptable through COM, you will not need other documentation. The documentation is available at <http://msdn2.microsoft.com/en-us/library/ms533050.aspx>.

To test our abilities to manipulate the HTML DOM, we will create a new script called `Get-WebSearchResults.ps1`. It is based on the previous example, but this time, it does not show any windows to the user. It silently navigates to the Live.com search page, submits a query, and then extracts the links to found pages from the search results. The script relies on the fact that Live.com generates a `<div>` HTML element with an ID of "results" that contains a bunch of links pointing to the pages that we need. Links are `<a>` elements that have a `href` attribute containing the URL we need. Here is the code of our script:

```
function WaitForLoad ($ie)
{
    while ($ie.Busy)
    {
        sleep -milliseconds 50
    }
}

$ie = New-Object -COM InternetExplorer.Application
$ie.Navigate("http://www.live.com")
WaitForLoad($ie)

$searchBox = $ie.Document.getElementById("q")
$searchBox.value = "`"Pro Windows PowerShell`""
$goButton = $ie.Document.getElementById("go")
$goButton.click();

WaitForLoad($ie)

$resultsDiv = $ie.Document.getElementById("results")
$links = [object[]] $resultsDiv.getElementsByTagName("a")
$realLinks = $links | where { $_.innerText -ne "Cached page" }
$realLinks | select innerText,href | Format-List

$ie.Quit()
```

The `getElementsByTagName()` method returns all child elements that have the specified tag; we use that to get all links inside the "results" element. We filter out the links that contain the text "Cached page"—we do not need those, as they point to the cached versions of the pages we are after. Once we have the links we need, we get just their `innerText` and `href` properties and pipe them through the `Format-List` cmdlet. Here is what we get when we run our script:


```
PS> .\Get-WebSearchResults.ps1
```

```
innerText : Bookpool: Pro Windows PowerShell  
href      : http://www.bookpool.com/sm/1590599403
```

```
innerText : APRESS.COM : Pro Windows PowerShell : 9781590599402  
href      : http://www.apress.com/book/view/1590599403
```

```
innerText : Getting Started With Windows PowerShell  
href      : http://www.microsoft.com/technet/scriptcenter/topics/winps  
           h/manual/start.mspix
```

```
innerText : Running Windows PowerShell Scripts  
href      : http://www.microsoft.com/technet/scriptcenter/topics/winps  
           h/manual/run.mspix
```

```
...
```

Windows Script Host Code Interoperability

PowerShell and .NET are really cool technologies, but it will take time until they take over the entire world. There is a lot of useful code out there written in VBScript, JScript, or other Windows Script Host-compatible languages. It would be very unwise, and maybe downright impossible, to throw all that code away and start rewriting everything in PowerShell. Besides existing code, the Web is full of examples automating various programs all written in VBScript or JScript. Sometimes, the examples are not short, and we cannot convert them to PowerShell on the fly. We need a usable way for our PowerShell code to interoperate with code written in JScript and VBScript.

Luckily, we have a good tool that can help us here—the `MSScriptControl` COM object available through the `MSScriptControl.ScriptControl` ProgID. It has been designed so that it can host a script environment; we can register code with it and later execute that code. We can use this object as a way to load an existing VBScript or JScript file from PowerShell. Once loaded, we can call functions inside those files, pass parameters, and get return values.

To demonstrate how the script control object works, we will work with two script files that know how to calculate the size of a file in bytes. One of them, `FileSize.js`, is written in JScript, and the other, `FileSize.vbs`, in VBScript. Both files define a function, `GetFileSize()`, that accepts a path to a file and returns a nicely formatted string containing the path to the file and the file size. Here is the JScript implementation:

```
function GetFileSize(filePath)  
{  
    var fileSystem = new ActiveXObject("Scripting.FileSystemObject");  
    var file = fileSystem.GetFile(filePath);  
    return (filePath + " has " + file.Size + " bytes.");  
}
```

And here is the VBScript one:

```

Function GetFileSize(filePath)
    Dim fileSystem
    Set fileSystem = CreateObject("Scripting.FileSystemObject")

    Dim file
    Set file = fileSystem.GetFile(filePath)
    GetFileSize = filePath & " has " & file.Size & " bytes."
End Function

```

As you can see, each language has its quirks in declaring functions and variables and working with COM objects.

To be able to invoke the JScript function in PowerShell, we need to read the `FileSize.js` file contents and pass them to the script control `AddCode()` method. That method will evaluate the function definition and make it available for later use.

Note It is best to load script files that contain only function definitions inside the script control. The `AddCode()` method evaluates code that has been given to it, and it will execute any code that has been defined outside a function definition. That can lead to unexpected results and undesired behavior. Your script files may need some modifications so that code is properly separated in functions before it can be loaded in the script control. The best way to look at such script files is as if they are script libraries that expose functions and make them available to PowerShell code.

Evaluating Code

After we have our script code included in the script control, we can call the functions defined in JScript. To do that, we can use either the `Eval()` method or the `Run()` one. The `Eval()` method takes a string as its input and executes it. When it is done, it returns the result to the caller—the PowerShell script in our case. The `Run()` method is more structured. It accepts several parameters: the first one is a function name, and the rest are parameters that will be passed to the function. The method returns the function return value to the PowerShell script. To see everything in action, we will create a script, called `JScriptEval.ps1`, which will call the JScript function using both `Eval()` and `Run()`. Here is the code:

```

$jscript = New-Object -COM MSScriptControl.ScriptControl
$jscript.Language = "JScript"
$jslines = Get-Content "FileSize.js"
$jsCode = [string]::Join("`n", $jslines)
$jscript.AddCode($jsCode)
$fileName = (dir FileSize.js).FullName

Write-Host "Using Eval"
$jscript.Eval("GetFileSize(`"$($fileName.Replace('\', '\\'))`")")

Write-Host "Using Run"
$jscript.Run("GetFileSize", $fileName)

```

The code starts with an important step in setting up the script control object—by setting the `Language` property to “JScript”, it tells it to interpret all code as JScript. It then uses the `Get-Content` cmdlet to get all the lines inside the `FileSize.js` file. The code next assembles the resulting script code by joining the lines with a new line symbol in between and passes that to `AddCode()`.

Note the huge difference between calling `Eval()` and `Run()`. `Eval()` needs to escape the double quotes to generate a JScript string literal. Additionally, it has to replace backslashes with double backslashes, because the backslash is a special escape character in JScript, and it needs to be escaped in string literals. The `Run()` method call is a lot better; we do not need to escape anything, because we use the script control facilities for passing parameters. Here is what happens when we run our script:

```
PS> .\JScriptEval.ps1
Using Eval
Z:\19 - Talking to COM Objects\FileSize.js has 211 bytes.
Using Run
Z:\19 - Talking to COM Objects\FileSize.js has 211 bytes.
PS>
```

The results from the `Eval()` and `Run()` methods are identical, and it is always a better idea to use the `Run()` one and spare yourself a headache or two.

Let’s now see how things work on the VBScript side. We will create a script, similar to the preceding one that calls the `GetFileSize()` defined in VBScript code. We will call it `VBScriptEval.ps1`; here is the code:

```
$vbscript = New-Object -COM MSScriptControl.ScriptControl
$vbscript.Language = "VBScript"
$vbsLines = Get-Content "FileSize.vbs"
$vbsCode = [string]::Join("`n", $vbsLines)
$vbscript.AddCode($vbsCode)
$fileName = (dir FileSize.vbs).FullName

Write-Host "Using Eval"
$vbscript.Eval("GetFileSize(`"$fileName`")")

Write-Host "Using Run"
$vbscript.Run("GetFileSize", $fileName)
```

It is pretty similar to the JScript version you saw before. The biggest difference is the `Language` property value that we set to the script control object. A small thing to note here is that we do not need to escape backslash characters in our path when we call `Eval()` here, because backslashes have no special meaning in VBScript. Here is what we get when we run the script:

```
PS> .\VBScriptEval.ps1
Using Eval
Z:\19 - Talking to COM Objects\FileSize.vbs has 258 bytes.
Using Run
Z:\19 - Talking to COM Objects\FileSize.vbs has 258 bytes.
PS>
```

Exposing Objects from MSScriptControl

The `Eval()` and `Run()` methods do the job of providing access to foreign functions quite well. They still feel like a bad hack, because they do not behave like real methods. We can do better if we exploit a feature of the script control object—dynamic object generation. The `MSScriptControl` object has a `CodeObject` property that returns an anonymous object composed of the scripts added to the script control. The code object will expose all functions that have been added as public methods. This allows us to have some nice syntax that practically makes foreign script objects indistinguishable from regular .NET objects.

We can use that to call our JScript function. We will create a new script file, `JScriptCodeObject.ps1`, which will again configure a script object, but this time will call its functions through the dynamically generated object. Here is the code:

```
$jscript = New-Object -COM MSScriptControl.ScriptControl
$jscript.Language = "JScript"
$jsLines = Get-Content "FileSize.js"
$jsCode = [string]::Join("`n", $jsLines)
$jscript.AddCode($jsCode)
$fileName = (dir FileSize.js).FullName

$fileSize = $jscript.CodeObject
$fileSize.GetFileSize($fileName)
```

The syntax is much friendlier now. Let's see what we get when we run the script:

```
PS> .\JScriptCodeObject.ps1
Z:\19 - Talking to COM Objects\FileSize.js has 211 bytes.
PS>
```

Let's now compare that to the VBScript version. Here is the code for the `VBScriptCodeObject.ps1` script that does the same thing for our VBScript code:

```
$vbscript = New-Object -COM MSScriptControl.ScriptControl
$vbscript.Language = "VBScript"
$vbsLines = Get-Content "FileSize.vbs"
$vbsCode = [string]::Join("`n", $vbsLines)
$vbscript.AddCode($vbsCode)
$fileName = (dir FileSize.vbs).FullName

$fileSize = $vbscript.CodeObject
$fileSize.GetFileSize($fileName)
```

There's not much of a difference from the JScript version. Here is what happens when we run it:

```
PS> .\VBScriptCodeObject.ps1
Z:\19 - Talking to COM Objects\FileSize.vbs has 258 bytes.
PS>
```

Summary

COM scripting can be both a very rewarding and a very frustrating experience. PowerShell provides excellent support for working with COM objects, but as you have seen from the examples in this chapter, hidden things still may come out to bite you when you least expect them. Hopefully, the examples we have gone through have covered the major interoperability problems, like working with collections and indexer properties, so that you know and fix them when you see them. What is left now is to explore the various applications and parts of the operating system that you use in your daily work. Chances are, those applications and services already have a COM automation interface exposed that is waiting for you to take advantage of it.



Managing Windows with WMI

Managing a large heterogeneous network has always been a formidable task. It is easy to manage a single program or a service running on the local machine using various automation interfaces such as PowerShell cmdlets, COM objects, or even .NET-based programmability. We get to the hard part when we throw distributed operations in the mix. What if the program we need to manage runs on a computer behind a firewall? What if it is not a program at all? It may be a hardware device such as a network router. This is where Windows Management Instrumentation (WMI) can help us. WMI is a set of technologies that work together to provide uniform access to objects on a network. We can use it to query services for information, extract all sorts of data, trigger commands, and change configuration options.

WMI objects are special types of objects that are exposed to Windows programs as COM objects. They are not regular COM objects in the sense that they provide standard interfaces and services built on top of COM that we can use. The PowerShell designers have recognized the important part WMI plays in Windows administration, and since PowerShell is primarily an administration tool, they have taken extra care to make working with WMI objects as easy as possible.

A Brief History of WMI

By 1996, a number of companies had recognized the need for a new approach to systems management that offered better standardization and tighter integration among components. The most popular standard management protocol at the time, Simple Network Management Protocol (SNMP), had proven inadequate to meet all the management needs of large enterprises. It is, at its essence, simple, which means that it was targeted at managing network devices, such as routers. Conceived at the end of the 1980s, the protocol had to support devices with little computational power, and thus it had to be extremely lightweight. That requirement unavoidably limited the functionality it could offer. Another thing that needs mentioning is that being only a protocol, SNMP cannot offer the full management support that the typical enterprise system needs. At minimum, we need central object repositories that allow us to discover objects on the network and scripting support.

Those shortcomings led to several companies, including Microsoft Corporation, Cisco Systems, and Intel Corporation, getting together and starting the Web-Based Enterprise Management (WBEM) initiative. It was an effort of providing a complete standard infrastructure for managing all components in a system. The Distributed Management Task Force (DMTF) consortium has subsequently taken ownership of the WBEM initiative and its standardization and marketing. At this moment, we have several implementations of the WBEM infrastructure

and the standards that the DMTF has defined are supported by many operating system, programs, and hardware devices. The most complete implementation of the standard by far has been Microsoft's—yes, WMI is an implementation of the WBEM standard.

Microsoft has chosen WMI as the strategic technology for supporting the Windows administration model. The first version of WMI saw the light of day with the Windows 98 and Windows 2000 releases. The technology has been deeply integrated in the operating system kernel, and anyone can use it to obtain information on just about any operating system, hardware device, or a piece of third-party software. The only requirement is that the managed object is running on or is connected to a Windows system and there exists a WMI provider that supports it.

WBEM and WMI Components

I have been referring to WBEM as a set of standards and an approach to systems management. To fully satisfy the management needs of the distributed enterprise environment, we need more than just a network protocol specification, so the DMTF folks have defined several components that are the building blocks for WBEM and WMI.

The Common Information Model

The ultimate goal of any standardization effort is to provide a uniform way of accessing all objects on the network. “Objects” typically means components in a software system, but WBEM goes even further by including hardware devices in that category. After all, there is no point in configuring a service running on one machine differently than you’d configure a hardware device driver or even a network router. To make that standardization possible, we need a way to define an object and its properties and operations. This is what the Common Information Model (CIM) does for us. The CIM standard defines an object-oriented information model that describes all manageable entities as objects that have a set of properties and operations. The CIM is based on the Unified Modeling Language (UML) that is the de facto standard for describing an object-oriented system. Being so close to UML makes CIM familiar to people with an object-oriented background, as it uses the same concepts present in .NET and other object-oriented systems. Object operations are defined by classes, and classes are cleanly separated in namespaces. Not only that, classes can inherit from other classes, and that allows easy customization or extension of existing classes.

Note A WMI object always belongs to a class. A class is, in short, the object type. Classes define how objects behave—what their properties and methods are. WMI classes are different than .NET classes, though. We cannot use the `GetType()` method to distinguish between different types of WMI objects; instead, we have to use their special `__CLASS` property. See the “WMI System Properties” section for details on getting information about a WMI object’s type.

CIM classes are provided and implemented by special providers that have been registered with the system. In the case of WMI, providers are COM DLLs that have been registered with the system registry. This architecture allows for easy extensibility. A program or service can

expose a management interface by creating a provider. That provider will register additional classes in a different namespace, but other than that, administrators will be able to manage that piece of software just as they would manage any other WMI component.

The CIM standard defines a set of standard classes, such as `CIM_Process`, that must be present on any system. It is good knowing about those and using them where possible, but that may not be sufficient for our needs at all times. The standard cannot possibly describe everything about an object and yet be as generic as possible. For example, a Windows process is a bit different than a process running on a UNIX system. There are management operations for a Windows process that would make no sense in a UNIX context. The WMI designers at Microsoft have solved this problem by extending the base CIM classes and using them wherever possible. To continue with the process example, WMI offers a `Win32_Process` class that inherits from and, therefore, extends the `CIM_Process` one. A `Win32_Process` object is still a valid `CIM_Process` object and supports all operations a `CIM_Process` would support. The inheriting class adds to the base properties; `Win32_Process` adds several Windows-specific properties, such as `ExecutablePath` and `ThreadCount`.

Network Transport

Object and information models are only a piece of the entire management puzzle. We need to be able to talk to objects anywhere on the network, and to be able to do that, we need a standard protocol. One of the first attempts to specify a standard was DMTF's CIM-XML protocol, which specifies object operations in XML and uses HTTP to transport the XML messages. For one reason or another, Microsoft did not choose to implement CIM-XML for WMI. WMI uses the Distributed COM (DCOM) infrastructure to connect to objects running on the network. That approach has the benefit of integrating with an existing infrastructure on Windows systems. Unfortunately, it also makes WMI incompatible with other WBEM implementations at the network-protocol level. For example, a Java client running on Solaris will only be able to query its own WBEM implementation, and there is no way for it to touch a WMI object that is a part of a Windows network.

Recently, the DMTF have defined another WBEM communications protocol: Web Services for Management, or WS-Management. It uses SOAP web services running over HTTP to encode all management operations. The WS-Management protocol has been developed as a way to provide a uniform protocol for managing all types of devices. The greatest benefit of using SOAP web services is that they are already supported by many languages and programming platforms, so implementing a management client should not be a hurdle. In the previous scenario, the Java client would have no problem connecting to Windows objects. Microsoft already has an implementation of the WS-Management standard in production; it is called Windows Remote Management (WinRM). It is bundled with Windows Vista, and you can install it as an add-on for previous versions of Windows. While we will be using the DCOM-based WMI protocol in this chapter, it may be useful to know that you can access the same type of WMI objects via WinRM.

Object Discovery

An important part of working with objects in a distributed environment is locating them. Knowing that you want to locate a `notepad.exe` process and terminate it is not enough; you need to know the machine that the process is running on and, if the machine is running several `notepad.exe`

processes, the process ID of the victim. WMI can use monikers, or paths that uniquely identify an object on the network. We can use two kinds of paths: relative and absolute. Relative paths identify objects running on the current computer and look like `Win32_Process.Handle="2824"` or `Win32_Service.Name="Dnscache"` for the process with the 2824 ID and the DNS caching service, respectively. Absolute paths look similar, and they just add the full machine name and the class namespace before the relative portion. The absolute paths corresponding to the relative ones that we saw previously look like `\\MACHINENAME\root\cimv2:Win32_Process.Handle="2824"` and `\\MACHINENAME\root\cimv2:Win32_Service.Name="Dnscache"`, respectively. Having an absolute path means that we can get a reference to the target object whether or not we run our client code on the same machine.

Another way to discover and access objects is to use object queries. WBEM defines a standard query language—the CIM Query Language, which is a subset of the SQL language used to query relational databases. WMI has a different query language, again based on SQL—the WMI Query Language (WQL). The good news is that CIM Query Language and WMI are very similar. Both support only data retrieval and lack SQL's data modification features like the `UPDATE`, `DELETE`, and `INSERT` statements. In this chapter, we will be working with WQL. The query language makes classes look like database tables. Object instances are represented by rows and properties define columns. Each object exposes its property values as cell values for the corresponding column definitions. To get a feeling of what a query looks like, let's look at a query that returns all processes running on the system. The WQL code looks like `SELECT * FROM Win32_Process`. That query requests all instances of the `Win32_Process` class, and the asterisk (*) is a shorthand notation that specifies that all available properties should be returned. We can specify a subset of properties and, in addition, filter the results; here is the query that will return the name and priority for all processes that start with the letter "A":

```
SELECT Name,Priority FROM Win32_Process WHERE Name LIKE 'A%'
```

The `LIKE` operator in our query uses the `%` symbol to denote zero or more repetitions of any character.

PowerShell's WMI Support

Now that we have had all the theory behind us, we can move to something real and tangible. What does it take to work with WMI objects from within PowerShell? The standard approach before PowerShell was to use the scripting API exposed as a set of COM objects. While writing COM client code is possible in every language, the typical WMI client code was written in VBScript, as that was easiest to do. The code involved getting an instance of the CIM object manager and giving it a WQL query. The result was a collection that was enumerated using a standard `For Each` loop that works in the same way as the `foreach` loop does in PowerShell. Just to get a taste of what the code is like, let's take a quick look at the `winword_pid.vbs` script that will get my `winword.exe` processes' IDs:

```
strComputer = "."
classNameSpace = "winmgmts:\\\" & strComputer & "\\root\\cimv2"
Set objWMIService = GetObject(classNameSpace)
```

```

query = "SELECT * from Win32_Process WHERE Name = 'winword.exe'"
Set results = objWMIService.ExecQuery(query)
For Each process in results
    Wscript.Echo "ProcessId: " & process.ProcessId
Next

```

Here is what running the script looks like when we use the Windows Script Host console application host, `cscript.exe`:

```

PS> cscript .\winword_pid.vbs
Microsoft (R) Windows Script Host Version 5.7
Copyright (C) Microsoft Corporation. All rights reserved.

```

```
ProcessId: 256
```

Voilà! I have only one instance of Microsoft Word running on my machine. What bothers me, and probably you too, is that, although the code gets the job done, it looks ugly. It is too verbose! I cannot believe we need eight lines of code to get a single object. Hang on—PowerShell does far better.

Get-WmiObject: The WMI Query Tool

The entry point for all WMI queries in PowerShell is the `Get-WmiObject` cmdlet. It takes a query and sends it to the CIM object manager. The result is a PowerShell collection that contains zero or more `System.Management.ManagementObject` objects. Those objects are the .NET wrappers for the CIM objects that we get back from WMI.

The VBScript code you saw in the preceding example is too verbose, because the COM objects we work with lack good default values for their parameters. The `Get-WmiObject` creators have recognized that as a problem and have chosen meaningful defaults. For example, we do not need to specify the machine name; if we do not, the cmdlet assumes we want an object running on the local machine. We do not need a namespace too; the default points to the most popular `root\cimv2` namespace. That namespace is likely to hold most of the objects we will ever need to manage: files, processes, services, network settings, device configurations—it is all there. Now, using the defaults, let's see how the `Get-WmiObject` version of the example we saw previously looks. We will create a new script, `winword_pid.ps1`, that does the same thing. Here is the code:

```

$query = "SELECT * from Win32_Process WHERE Name = 'winword.exe'"
Get-WmiObject -query $query | select ProcessId

```

The eight lines we needed in VBScript have gone down to two. Apart from relying on the better defaults, we use a PowerShell idiom to avoid looping through the result collection. Selecting and displaying a property value is just a matter of piping the result through the `select` cmdlet. Here is the output we get after running the script:

```
PS> .\winword_pid.ps1
```

```

ProcessId
-----
256

```

The preceding code uses the `-query` parameter to pass a fully specified WQL query. The WQL language `SELECT` statement can specify the properties we are interested in. We can use that to get a subset of an object's properties. That feature of the language has been meant to provide a way to reduce the amount of data that gets sent over the network in cases where we may be retrieving large collections of objects from remote machines. Our example runs on the local machine and specifies an asterisk (*) as the property list (the asterisk is shorthand specifying all properties). We do not need to optimize network traffic for local queries, and `Get-WmiObject` allows us to skip that part entirely by specifying the WMI class whose instances we want to retrieve. In the class case, we need to provide a filter string that is just the `WHERE` clause of our query without the "WHERE" string. We can shorten our command even further by using the convenient `gwmi` alias for the cmdlet. Here is how a command that gets the `winword.exe` process instance looks now:

```
PS> gwmi -class Win32_Process -filter "Name = 'winword.exe'"
```

```
ProcessName      : WINWORD.EXE
Handles          : 397
VM               : 369500160
WS               : 37044224
...
```

We can use our `Get-WmiObject` to query objects running on remote machines. To do that, we need to provide the `-computer` parameter. Accessing objects on another computer immediately raises questions about security. By default, the cmdlet will use our Windows domain credentials, but we can override that by providing the `-credentials` parameter. We can pass a real `PSCredential` object that we can obtain using the `Get-Credential` cmdlet. The cmdlet will get the credential by popping up a dialog before the user asking for a user name and password. There is an easier way—we can provide a user name string, and `Get-WmiObject` will ask for the password using the same credentials dialog. Here is how to use that to get an instance of the DNS service running on our `WIN2K3-TEST` machine:

```
PS> gwmi -class Win32_Service -filter "Name = 'DNS'" -computer WIN2K3-TEST `
-credential Administrator
```

```
ExitCode : 0
Name     : DNS
ProcessId : 1300
StartMode : Auto
State    : Running
Status   : OK
```

The preceding command will display the credentials dialog shown in Figure 20-1. If the user name and password are correct, we will get back an instance of our service.



Figure 20-1. *The PowerShell credentials dialog*

Language Support for WMI Objects

Before the second release candidate build's release, the only WMI support in PowerShell was provided by the `Get-WmiObject` cmdlet. Luckily, user feedback made the PowerShell team reconsider that. At the last possible moment, they added several language enhancements that simplify working with WMI objects. The most important enhancements are the ability to cast specially formatted strings into WMI objects. This allows us to use strings as WMI object monikers similar to URLs. The thing to keep in mind is that all casts that have been added are just convenience features. Everything they do can be replicated using `Get-WmiObject` and some knowledge about how WMI and CIM work.

The first convenience feature we are going to look into is the ability to convert strings containing WQL queries into WMI searcher objects. The type literal for that is `[wmisearcher]`—it will yield a `ManagementObjectSearcher` object. The only use we have for a searcher object is calling its `Get()` method. That will execute the WMI query and will return any matching objects. Returning to our `winword.exe` process example, here is how we can write it using a searcher object:

```
PS> $query = "SELECT * from Win32_Process WHERE Name = 'winword.exe'"
PS> $searcher = [wmisearcher] $query
PS> $searcher.Get()
```

```
ProcessName      : WINWORD.EXE
Handles          : 474
VM               : 400248832
WS               : 31821824
...
```

Using WMI searchers over calling `Get-WmiObject` directly is purely a matter of personal taste. There is no advantage to using one approach over the other when we need local objects. The WMI searcher approach gets unwieldy when we want to retrieve objects from remote machines. It might be possible to dig into the searcher object `Scope` property and configure the proper settings for a remote connection, like the endpoint address, the CIM namespace, and

the credentials, but that seems too hard. Calling `Get-WmiObject` and passing the `-computer` and `-credential` parameters is much simpler.

Maybe the most useful cast is the one that converts an object path to a real object. It is very convenient and saves us the necessity of storing unique parameters like process IDs or service names in order to run a WQL query just to get to the real object at a later point in time. We can use the `[wmi]` type literal and cast a relative or absolute object path. Here is how to obtain a reference to the DNS cache service using both an absolute and a relative path:

```
PS> [wmi] '\\TESTMACHINE\root\cimv2:Win32_Service.Name="Dnscache"'
```

```
ExitCode : 0
Name      : Dnscache
ProcessId : 1276
StartMode : Auto
State     : Running
Status    : OK
```

```
PS> [wmi] 'Win32_Service.Name="Dnscache"'
```

```
ExitCode : 0
Name      : Dnscache
ProcessId : 1276
StartMode : Auto
State     : Running
Status    : OK
```

Using object paths to retrieve objects is really convenient as it is supported by all object types and works for remote machines (when using absolute paths, of course). The alternative, if we were using `Get-WmiObject`, would be to craft special WQL queries that would have to be different for different types of objects. For example, the query for a process object would not work for services, as processes use their `Handle` property as a path component while services use the `Name` property. The only scenario where it would be best to use `Get-WmiObject`, again, is when trying to connect to a remote machine with different credentials than the current one.

Until now, we have been working exclusively with WMI object instances. WMI allows us to get a reference to classes, too. PowerShell makes that easier by allowing us to cast a string containing a class's name into the correct WMI class. The type literal that we need to use is `[wmi:class]`. Getting a class allows us to call static methods, or methods that do not belong to an object. To demonstrate that, we can get the `Win32_Process` class and create a new process by calling its `Create()` static method. Here is how we can do that:

```
PS> $processClass = [wmi] "Win32_Process"
PS> $processClass.Create("notepad.exe")
```

```
__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 2
__DERIVATION     : {}
__SERVER         :
__NAMESPACE     :
__PATH           :
ProcessId        : 3476
ReturnValue       : 0
```

The method returns a `__PARAMETERS` object that contains the newly created process's process ID. We could use that information later to obtain a reference to the process. This is how to do that by crafting a path that uses the process ID as the `Handle` property value:

```
PS> [wmi] "Win32_Process.Handle=3476"
```

```
ProcessName      : notepad.exe
Handles          : 48
VM               : 57446400
WS               : 7352320
Path             : C:\Windows\system32\notepad.exe
...
```

Again, the WMI class type cast syntax is really convenient—until you have to use it on a remote machine. We need a way to get a reference to a class object using standard WQL and pass that to `Get-WmiObject`. Luckily, the common information model contains class metadata that we can use. We can get our class object by querying for `meta_class` object instances. This is how to get the `Win32_Process` class using this method:

```
PS> gwmi -class "meta_class" -filter "__THIS ISA 'Win32_Process'"
```

```
Win32_Process
```

The filter uses the `__THIS` special property that points to the class object. The `ISA` operator will return true if the current class is `Win32_Process` or inherits from `Win32_Process`.

Now that we know how to get hold of a WMI class using WQL, let's do it for a remote machine. To demonstrate that, we will create a new `notepad.exe` process on the `WIN2K3-TEST` machine:

```
PS> $w32Proc = gwmi -class "meta_class" -filter "__THIS ISA 'Win32_Process'" `
    -computer WIN2K3-TEST
PS> $w32Proc.Create("notepad.exe")
```

```
__GENUS          : 2
__CLASS           : __PARAMETERS
__SUPERCLASS     : 
__DYNASTY         : __PARAMETERS
__RELPATH         : 
__PROPERTY_COUNT  : 2
__DERIVATION     : {}
__SERVER         : 
__NAMESPACE      : 
__PATH           : 
ProcessId        : 2876
ReturnValue       : 0
```

The preceding command does not specify user credentials for logging in to the remote machine; that means we are using the credentials of our currently logged on user.

Note The process created on the remote computer will not have a visible window, as it does not interact with the desktop of the currently logged on user. Windowed applications that wait for a user action before exiting will hang forever and need to be terminated from the Task Manager. The bottom line is that you should be using the remote `Win32_Process` technique only for launching commands that exit immediately after they finish their jobs. Of course, you can also use this approach for launching long-running programs that may be servicing local or network requests by users or other programs.

Exploring CIM Classes and Objects

One of my favorite PowerShell features—being able to poke at objects in the interactive command prompt—is available with WMI objects too. You can get an object and interrogate it for new features that you have not used yet. You can discover other objects in the process that can help you do your job better. Not to mention that knowing how to look around the system will be of immense value when troubleshooting problems.

Getting a List of Classes and Namespaces

The previous section introduced the `meta_class` WMI class. Every WMI class is actually an instance of `meta_class`, and knowing that, we can retrieve a list of all classes in a namespace. All we need is to issue a query that gets all instances of a given class. Here is how to do it with `Get-WmiObject`:


```
PS> gwmi -class "meta_class"
```

```
__SystemClass          __thisNAMESPACE
__Provider             __Win32Provider
Win32_OsBaselineProvider  __ProviderRegistration
...
```

We get quite a long list with the base class name in the first column and the actual class in the second one. If we are looking at narrowing results down, we can use the full power of the WQL language and provide a filter. Here is how to get all classes whose names start with the Win32 string:

```
PS> gwmi -class "meta_class" -filter "__CLASS LIKE 'Win32%'"
```

```
Win32_PrivilegesStatus      Win32_JobObjectStatus
Win32_Trustee               Win32_ACE
Win32_SecurityDescriptor    Win32_ComputerSystemEvent
Win32_ComputerShutdownEvent Win32_IP4RouteTableEvent
Win32_SystemTrace           Win32_ProcessTrace
Win32_ProcessStartTrace     Win32_ProcessStopTrace
...
```

The Get-WmiObject cmdlet has a shortcut that can get all classes in a namespace. We can do that by supplying the -list switch. Here is how:

```
PS> gwmi -list
```

```
__SystemClass          __thisNAMESPACE
__Provider             __Win32Provider
Win32_OsBaselineProvider  __ProviderRegistration
...
```

There is a limitation to that approach, though; the -filter parameter does not work. If we need to filter the results using WQL, we have to use the approach that gets all meta_class instances, as shown in the preceding code.

All of the preceding queries worked with the default namespace, root\cimv2. To get classes from other namespaces, we need to provide the -namespace parameter.

How do we know the namespaces on the system and how do we explore them? Again, we need to get all instances of the __NAMESPACE class. Namespaces are hierarchical, and the root of the hierarchy is at the root namespace. This is how to get all namespaces below root:

```
PS> gwmi -namespace root -class "__NAMESPACE" | select Name
```

```
Name
----
subscription
DEFAULT
```

```

MicrosoftDfs
MSAPPS11
CIMV2
...
Microsoft
aspnet

```

Of course, getting namespaces below the ones we listed is a matter of changing the `-namespace` parameter. This is how to get the namespaces below `root\Microsoft`:

```
PS> gwmi -namespace root\Microsoft -class "__NAMESPACE" | select Name
```

```

Name
----
SqlServer
HomeNet

```

WMI System Properties

Every WMI object has several special properties that allow us to extract system-related information. The WMI documentation calls those “system properties.” We have already introduced some of them in this chapter; they are the strange properties with names starting with double underscores, `__CLASS` for example. We can use them to get additional information about an object. Here is what they do:

- `__CLASS`: This property holds the name of the object’s class. We can use it to verify if a query has returned objects of the class that we expect.
- `__SUPERCLASS`: This property holds the name of the name of the parent class.
- `__DYNASTY`: This one contains the top-level class from which the object’s class and all its superclasses eventually derive.
- `__DERIVATION`: This is a collection of all the classes that the object’s class derives from. The first element is the `__SUPERCLASS` value, and the last one, the `__DYNASTY` class.
- `__NAMESPACE`: This holds a string with the namespace that contains the object’s class.
- `__GENUS`: This property is used to distinguish between classes and objects. Class objects have a value of 1 while objects contain a 2.
- `__SERVER`: This one contains the name of the machine that hosts the object.
- `__PATH`: This property holds the absolute, machine-qualified path to the object.
- `__RELPATH`: This one is the relative path to the object. It is valid only if the object and its client run on the same machine.
- `__PROPERTY_COUNT`: This property contains the number of nonsystem properties that the object has.

Querying Hardware Devices

The examples you have seen so far have dealt with software entities only: processes, services, and so on. WMI makes no distinction between hardware and software objects, and its tight integration with the Windows kernel allows it to manipulate all sorts of devices. We can use it to get information about different computers and their hardware configurations.

For example, we can get information about the currently installed hard drives by querying the available Win32_DiskDrive objects. Here is what I get on my system:

```
PS> gwmi -class Win32_DiskDrive
```

```
Partitions : 2
DeviceID   : \\.\PHYSICALDRIVE0
Model      : ST3320620AS ATA Device
Size       : 320070320640
Caption    : ST3320620AS ATA Device
```

```
Partitions : 1
DeviceID    : \\.\PHYSICALDRIVE1
Model       : USB FLASH DRIVE USB Device
Size        : 2056320000
Caption     : USB FLASH DRIVE USB Device
```

I get a list of all drives, including my USB flash drive that is plugged in at the moment. The preceding list contains physical drives only. We can also get information about the partitions defined for the physical disks by using the Win32_DiskPartition class. For example, we can get the boot partition on the system by filtering by the BootPartition property:

```
PS> gwmi -class Win32_DiskPartition -filter "BootPartition = True"
```

```
NumberOfBlocks : 104857600
BootPartition   : True
Name            : Disk #0, Partition #0
PrimaryPartition : True
Size            : 53687091200
Index           : 0
```

Partitions and mapped network drives get registered as logical disks and are usually assigned a drive letter. If we wanted to get the logical drives, we would have to use the Win32_LogicalDisk class instances. Here is how to get a list of all drives on the system:

```
PS> gwmi -class Win32_LogicalDisk
```

```
DeviceID       : C:
DriveType      : 3
ProviderName   :
```

```

FreeSpace      : 33011838976
Size           : 53687087104
VolumeName     :

DeviceID       : D:
DriveType      : 3
ProviderName   :
FreeSpace      : 144486309888
Size           : 266383388672
VolumeName     :

...

```

We can combine all drive information with information about the system BIOS, CPU, and memory by querying the `Win32_BIOS`, `Win32_Processor`, and `Win32_PhysicalMemory` objects. As you can see, creating a script that visits a number of computers on the network and extracts information about their hardware configurations should not be hard to do. We could use the gathered data for generating reports, detecting malfunctioning devices, and maybe even singling out computers that need hardware upgrades.

Getting Information About Software

An important regular maintenance task is keeping all our software up to date. Of course, the most important piece of software is the operating system. We can use WMI to query the state of installed service packs and updates for all computers on the network and then trigger alarms when something is not up to date, just like we could for hardware queries.

Operating System Updates

The first thing that we may need to check is the operating system version and its installed service packs. We can get that from the `Win32_OperatingSystem` object for the current machine:

```

PS> gwmi Win32_OperatingSystem | `
select Name,Version,BuildNumber,`
ServicePackMajorVersion,ServicePackMinorVersion | Format-List

Name                        : Microsoft® Windows Vista™ Ultimate |C:\Windo
                             ws|\Device\Harddisk0\Partition1
Version                    : 6.0.6000
BuildNumber                : 6000
ServicePackMajorVersion    : 0
ServicePackMinorVersion    : 0

```

Here is how that looks for a Windows XP system running service pack 2:

```

PS> gwmi Win32_OperatingSystem | `
select Name,Version,BuildNumber,`
ServicePackMajorVersion,ServicePackMinorVersion | Format-List

```

```

Name                : Microsoft Windows XP Professional|C:\WINDOWS
                    |\Device\Harddisk0\Partition1
Version             : 5.1.2600
BuildNumber         : 2600
ServicePackMajorVersion : 2
ServicePackMinorVersion : 0

```

Hot fix updates are also known as quick fix engineering updates. We can get information about those by getting the `Win32_QuickFixEngineering` objects on the system. Normally, there are quite a lot of those, as problems and vulnerabilities are discovered all the time and fixes are pushed through the Windows Update service. We can use the `HotFixID` property to get a list containing only the quick fix IDs:

```
PS> gwmi Win32_QuickFixEngineering | select HotFixID
```

```

HotFixID
-----
{5E5BD655-7AA9-47F9-BB6D-A1D8CE29AC86}
{C7A78F7F-EF32-4477-BAD7-3439EA7571BF}
{375F080F-88E9-4EA1-A177-C0F091546AC8}
{C6F1E87D-F3E1-4874-97EC-F87DAB6D6878}
{1DE969A4-D024-4CC3-AAC8-2B79C2751031}
{BE2BFE6C-42E9-41DC-925D-2B5AA1A78644}
933246
928439
932926
KB905866
KB925902
KB928253
KB929399
...

```

Installed Programs

Microsoft has done a great job of standardizing the program installation process through their Windows Installer technology. For quite some time, they have been requiring that any program that wants to meet the “Designed for Windows” logo requirement must have a Windows Installer-based installation. Those requirements are aimed at making administration of installed programs easier. Windows Installer packages define products; products are split into features, and features, in turn, are split in components. Features are parts of the product that the user may choose to install or skip during the installation process. Just like many other parts of the operating system, we can get information about the installed products and features using WMI. That makes the task of maintaining and upgrading installations of programs on many computers across the network significantly easier.

We can get a list of installed products on our system by getting all `Win32_Product` instances. It is likely that we will get a huge list, so it might be a good idea to use a WQL filter to get just the products we are interested in. Here is how we can get all products whose names start with “PowerShell”:

```
PS> gwmi Win32_Product -filter "Name LIKE 'PowerShell%'"
```

```
IdentifyingNumber : {2863EE03-3DE6-4816-BBFC-5B8A59DCF743}
Name               : PowerShell Community Extensions 1.1.1
Vendor             : PowerShell Community Extensions Developers
Version            : 1.1.1
Caption            : PowerShell Community Extensions 1.1.1
```

I get only an entry about the PowerShell Community Extensions. It is a free product that contains a set of additional cmdlets, providers, and scripts, which we will discuss in Chapter 21. You might be wondering why we did not get PowerShell itself. The answer is simple: Windows PowerShell is not considered a product. It is distributed as an add-on for the operating system, and for all purposes, we can think of it as a part of the operating system.

Note Running the code that gets instances of the Win32_Product class may sometimes get you a Get-WmiObject : Generic failure error on Windows Vista. The error is caused by a known bug in the Vista WMI implementation and will occur for any query for the Win32_Product objects, whether or not the client is PowerShell. The bug should be addressed for Windows Vista's first service pack release.

Many installation programs based on Windows Installer allow a user to choose which program features to install on the hard drive. Features can be installed or removed later. We can get all features or all features for a product using WMI, which makes checking if a feature is installed very easy. Here is how to get the captions and versions for all features of Microsoft Word that have been installed on the system:

```
PS> gwmi -class Win32_SoftwareFeature `
-filter "ProductName LIKE 'Microsoft Office Word%'" | `
select Caption,Version
```

Caption	Version
-----	-----
GIMME OnDemand Feature.	12.0.4518.1014
	12.0.4518.1014
	12.0.4518.1014
	12.0.4518.1014
Help	12.0.4518.1014
	12.0.4518.1014
	12.0.4518.1014
Bibliography Files	12.0.4518.1014
Document Parts Files	12.0.4518.1014
Word Templates	12.0.4518.1014
	12.0.4518.1014
Quick Formatting Files	12.0.4518.1014
Microsoft Office Word	12.0.4518.1014

The code uses a simple yet effective trick—a LIKE comparison on the `ProductName` property. You can use that approach for easy checks for products whose full names are hard to find or remember.

The Network Configuration

Network adapters are just another type of hardware with corresponding WMI objects that we can use to our advantage. To get information about the hardware device, we can query the `Win32_NetworkAdapter` objects on our system. Here is how the default view of my VIA Rhine II-compatible network adapter looks:

```
PS> gwmi Win32_NetworkAdapter -filter "Name LIKE 'VIA Rhine%'"
```

```
ServiceName      : FETNDIS
MACAddress       : 00:08:A1:93:10:9A
AdapterType      : Ethernet 802.3
DeviceID         : 4
Name             : VIA Rhine II Compatible Fast Ethernet Adapter
NetworkAddresses :
Speed            : 100000000
```

The information is strictly hardware related. You will not see anything related to the Internet, like an IP address assigned to the adapter. Those settings are available through the `Win32_NetworkAdapterConfiguration` objects. Here is the corresponding configuration object for my network adapter:

```
PS> gwmi Win32_NetworkAdapterConfiguration `
-filter "Description LIKE 'VIA Rhine%'"
```

```
DHCPEnabled      : True
IPAddress        : {85.11.181.215}
DefaultIPGateway : {85.11.181.1}
DNSDomain        : somedomain.net
ServiceName      : FETNDIS
Description      : VIA Rhine II Compatible Fast Ethernet Adapter
Index           : 4
```

Note that, this time, we are using a filter on the `Description` property. As you can see, we get the IP address, the default gateway, and other Internet-related settings.

Generating WMI Code

WMI has been around for quite some time. It has its quirks and difficulties, but it offers huge benefits to its users. The hardest part about WMI is the ocean of information that surrounds us. WMI's greatest strength—being comprehensive and covering all aspects of Windows administration—is also the biggest obstacle to people using it. There are too many classes to effectively

memorize, and their instances have so many properties that it is impossible to learn them all and start using them by hand. The only solution to this problem is to have some sort of a tool—either an integrated development environment or a code generator—that can help us in writing code. A number of free tools, available on the Internet, can enumerate all classes and their properties and then generate code that will query for object instances and access their properties. This can help tremendously in starting a new script that uses an unknown object. The only problem is that those tools do not support PowerShell yet. The best they can do is to generate VBScript code. Luckily, we can convert the generated code to PowerShell relatively easily. You do not need to know VBScript to get the general idea of creating the right query. You can then inspect the properties referenced by the generated code and just use the same properties from PowerShell.

Scriptomatic

The Scriptomatic tool was the first WMI scripting helper developed by Microsoft. It is hugely popular and has helped a great number of people. In a nutshell, it is a small utility that lists all WMI classes for you and generates code that gets all object instances and prints out their properties for the class you selected. The original version generated only VBScript code, but the newest version, Scriptomatic 2.0, can now generate code in several different languages: VBScript, JScript, Perl, and Python. Unfortunately, PowerShell is not available yet, so we will have to convert the generated code from VBScript by hand. As you will see in a second, that is not hard at all.

Note You can get Scriptomatic 2.0 from its home on the Microsoft TechNet site at <http://www.microsoft.com/technet/scriptcenter/tools/scripto2.mspx>. The tool is distributed as a self-extracting archive that contains a HTML application (.hta) file. You may get script errors when running the file if you do not do use elevated privileges on Windows Vista. To run the tool with elevated privileges, right-click the Internet Explorer icon, and select “Run as administrator” from the context menu. When the browser window loads, select Open from the File menu (press the Alt key on your keyboard to display the menu if it is not initially shown), navigate to the folder where you installed Scriptomatic, and open the ScriptomaticV2.hta file. Make sure you choose All Files as the File Open dialog filter, or you may not see the HTA file in the target folder. You will be prompted with a security warning dialog; click the Run button, and you are ready to go.

The Scriptomatic application window allows you to select a WMI class from a drop-down list. Once you do, the tool will generate code that gets all instances and prints their properties. You can run the generated code on the spot by clicking the Run button, and you can copy or save the code to another place for later use.

Let’s now use Scriptomatic to generate a script that gets details about monitor resolution. We need to select the Win32_DesktopMonitor class from the drop-down, as shown in Figure 20-2.

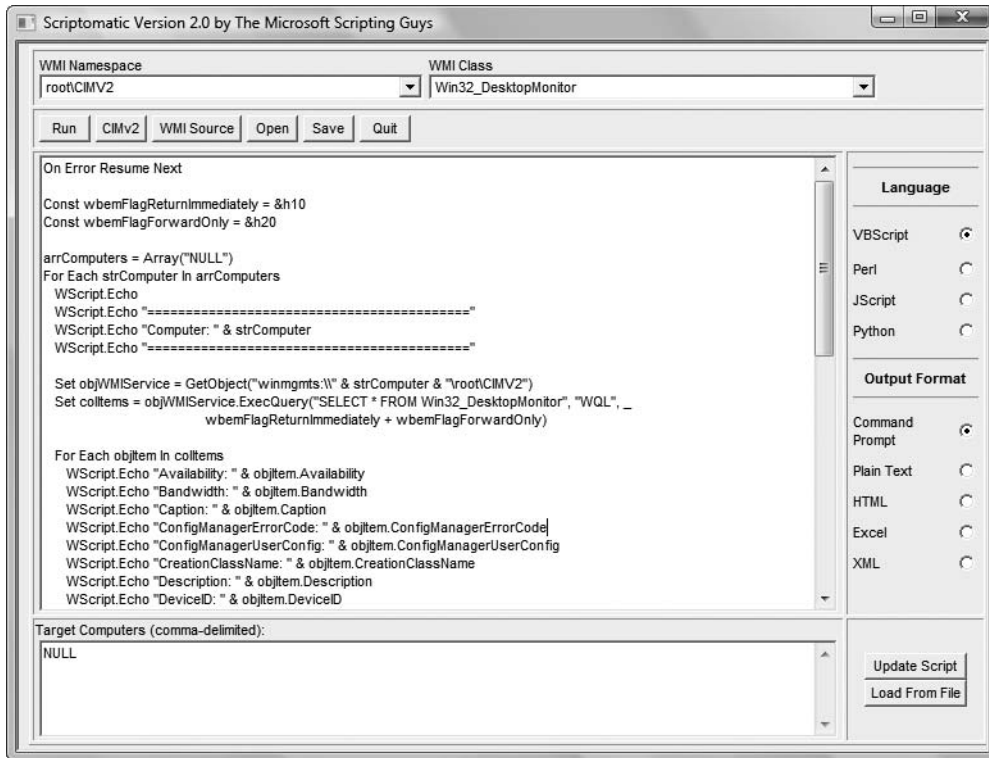


Figure 20-2. *The Scriptomatic window showing the generated code*

Do not run away yet! That is a lot of code, but we only need a small portion of it. The most important part is the WQL query. We simply need to copy the string with the SELECT statement and paste it as a query that gets sent to `Get-WmiObject`. That is all! Here is the result:

```
PS> gwmi -query "SELECT * FROM Win32_DesktopMonitor"
```

```
...
ScreenHeight           : 900
ScreenWidth             : 1440
...
```

We can narrow down the results by selecting only the properties we need. Initially, you might want to glance over the VBScript code and go over the property names, which may help you stumble upon the `ScreenHeight` and `ScreenWidth` properties. All you have to do to get those from PowerShell is pipe the `Get-WmiObject` result through `select`:

```
PS> gwmi -query "SELECT * FROM Win32_DesktopMonitor" | `
select ScreenHeight,ScreenWidth
```

ScreenHeight	ScreenWidth
-----	-----
900	1440

You may encounter properties that do not get displayed by PowerShell's default view. One such example is the `Win32_BIOS` class. The shell will not display some of its properties like `Caption` or `CurrentLanguage`. Here is the default view that we get:

```
PS> gwmi Win32_BIOS
```

```
SMBIOSBIOSVersion : 0401
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 02/17/06 01:14:37 Ver: 08.00.12
SerialNumber      : System Serial Number
Version           : DELL - 2000617
```

Code generated by Scriptomatic uses properties not included in the default view. After we find out about their names, we can force PowerShell to display them:

```
PS> gwmi Win32_BIOS | select Caption,CurrentLanguage
```

Caption	CurrentLanguage
-----	-----
BIOS Date: 02/17/06 01:14:37 Ver...	en US iso8859-1

Summary

WMI is a set of mature technologies that represent a powerful weapon in every administrator's daily fight against network entropy. With the advent of Windows Remote Management, WMI is becoming even more attractive, as we can use it without having to go through the burden of reconfiguring firewalls to allow DCOM traffic. If you have not used WMI for real work yet, now may be the best time to start doing so. Of all WMI-related scripting tools that I have used, PowerShell is the one that makes using, learning, and mastering WMI the easiest.

Learning WMI can be a long journey. It takes some time to master because of the sheer number of objects that can be scripted with it. You can get a lot of useful ideas from the Microsoft Script Center's script repository available at <http://www.microsoft.com/technet/scriptcenter/scripts/msh/>. It contains many sample scripts, most of which use WMI to script an operating system component. Another place to get ideas is the WMI classes reference that is available at <http://msdn2.microsoft.com/en-us/library/aa394554.aspx>. The reference contains full documentation on all the classes and is the place to look for help when you need to know what a specific property or method does. Finally, to get the big picture for web-based enterprise management, you can find the DMTF specifications at <http://www.dmtf.org/standards/wbem/>. Specifications can easily get boring, and you can get the essence of the standards from the DMTF tutorial that is available at <http://www.wbemsolutions.com/tutorials/DMTF/>.

Even though WMI is a pretty complex subject and requires some time to fully get your head around it, PowerShell offers very good support and really makes WMI objects first class citizens. Now that you do not have to deal with a number of obscure COM objects to get information from WMI, you can really start using this tool in your daily management tasks. In this chapter, we have used WMI mostly to query objects for information.

WMI also supports subscribing to specific events so that we can get notified whenever something interesting happens. Unfortunately, PowerShell does not support subscribing to those events in its current state. We can still subscribe and handle those events, but we need to use a free snap-in, PSEventing, which I will demonstrate in Chapter 22.



PowerShell Community Extensions

One of the wisest sayings about software development goes like this, “To release is to choose.” It stems from the idea that if you wait for release until you have a software product with all the features users might want, your development cycle will be endless; there will always be more and more ideas, better ways to implement something, a faster algorithm that will make a difference in some scenarios. You cannot do this forever. Well, you can, but the market will not wait for you. The product will share Duke Nukem Forever’s destiny—it will remain nothing more than vaporware. The PowerShell team heeded that lesson very well—there are a ton of useful commands and enhancements that never made it for the version 1.0 release. The team decided that what we have now is good enough and will provide a lot of value to users, even without some of the bells and whistles they could have added.

Now, the community has to step in. PowerShell is very extensible, and we can easily implement any missing utility as an extra script or as a snap-in that contains new cmdlets. This is how the PowerShell Community Extensions project got started. It tries to collect all the useful cmdlets, scripts, functions, and filters that members of the community are interested in. It really is the perfect companion for any new PowerShell installation. Whenever you find that something is missing in the default install, checking to see if the PowerShell Community Extensions project has it already is worth the time. Chances are, you’ll find what you’re looking for.

The best part about the project is that every piece of code is released under an open source license. Anyone can get the code and use it to learn how the utilities are implemented. Tinkering with the code is one of the best ways to learn new tricks. We learn not only how the code at hand works but about new services that the shell exposes. We can use those in a different context to get new ideas for doing our jobs easier.

Installation and Configuration

Enough praise—let’s get started! The PowerShell Community Extensions (PSCX) project is hosted on CodePlex, Microsoft’s open source project hosting web site. The project page is located at <http://www.codeplex.com/PowerShellCX>. Download the latest release, and install it. The installer, like any normal installation program, will require elevated privileges. The only special thing you have to decide at the time of installation is whether you would like to install the default profile script for the project. It is best if you do so; the profile script takes care to add the PSCX snap-in to your shell session every time that you start your shell. In addition,

it defines several important functions and variables. Do not worry about losing your original profile; it will be backed up to a file next to the new profile script, and you can easily include or dot source your original profile script in the new one.

Tab Expansion

PSCX comes with an improved version of the default tab expansion mechanism. In fact, it completely replaces the default tab expansion. You can refer to Chapter 11 for details on changing the way the shell expands strings when the user presses the Tab key. After installing the snap-in, we have a completely new `TabExpansion` function:

```
PS> (Get-Command TabExpansion).Definition
param($line, $lastWord) Get-TabExpansion $line $lastWord
```

The original PowerShell-supplied function is quite wordy, while it seems that PSCX delegates all expansions to the `Get-TabExpansion` cmdlet. The cmdlet can be called directly from code. We need to only set the `$line` and `$lastWord` parameters:

```
PS>Get-TabExpansion Win32_Op Win32_Op
Win32_OperatingSystem
Win32_OperatingSystemAutochkSetting
Win32_OperatingSystemQFE
```

From the preceding example, you can see one of the additions to the tab expansion mechanism—PowerShell can now complete WMI class names. Another new thing is completions for .NET type names. Why do we really need a cmdlet anyway, and what good is calling it on the command line? The answer lies in a typical extensibility scenario: suppose we have extended the default tab expansion mechanism by defining our own `TabExpansion` function, and we want both expansions to work according to our scheme and to the one provided by PSCX. The best way to do that is to combine our expansions with the ones returned by `Get-TabExpansion`. If you need to customize your tab expansion and need to call that cmdlet from your own code, please refer to Chapter 11.

Editing Configuration Files

UNIX shells have long since relied on the `$EDITOR` global variable to determine what is our preferred text editor so that they use it when we need to edit a file. With PSCX, we can now have this feature in PowerShell, too. If you are using the default PSCX profile script, you should have the `Edit-File` function available. It will use the currently configured editor to open the file it has been given. You can pass the file name as a parameter or pipe it in. In addition, the function has been aliased to `e`, so that we can open a file for editing in a blink:

```
PS> e test.txt
PS> $PscxTextEditorPreference
Notepad
PS> $PscxTextEditorPreference = 'C:\Program Files\NotepadPlusPlus\notepad++.exe'
PS> e test.txt
PS>
```

The default editor is `notepad.exe`. You can change it by setting the `$PscxTextEditorPreference` variable, preferably in your profile script. In the preceding example, I set it to my current favorite text editor—Notepad++.

PSCX recognizes the need to easily edit our profile scripts and provides four helper functions that feed those files to our favorite text editor:

- `Edit-GlobalHostProfile`: Edits the profile script for the current host application globally for all users
- `Edit-GlobalProfile`: Edits the global profile script that affects all users and all host applications
- `Edit-HostProfile`: Edits the current host profile script for the current user
- `Edit-Profile`: Edits the current user's all-hosts profile script

For a detailed explanation on shell hosts and the various ways you can configure your shell session through a shell profile script, refer to Chapter 11. The preceding four functions can be considered as a shortcut to performing the customizations that Chapter 11 describes.

Getting Help

How do we know what commands are available once we have installed PSCX? The “I have installed PSCX, now what?” problem is not uncommon, so several helper functions are available to point us in the right direction by showing a list of commands.

Probably the most important function here is `Get-PscxCmdlet`. It returns a (long) list of all the cmdlets that we have just installed:

```
PS> Get-PscxCmdlet
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Get-ADObject	Get-ADObject [-Domain <...]
Cmdlet	Resolve-Assembly	Resolve-Assembly [-Name...]
Cmdlet	Test-Assembly	Test-Assembly [-Path] <...]
Cmdlet	ConvertFrom-Base64	ConvertFrom-Base64 [-Ba...]
Cmdlet	ConvertTo-Base64	ConvertTo-Base64 [-Path...]
...		

The list is quite long—59 cmdlets in total for the version 1.1.1 release. I had to convert them to a comma-separated value list and use Microsoft Excel to keep track of them when I started writing this chapter. Here is how you can view the list in Excel (the code assumes that you have Microsoft Excel installed and it is associated with `.csv` files):

```
PS> Get-PscxCmdlet | Export-Csv cmdlets.csv
PS> Invoke-Item cmdlets.csv
PS>
```

Having a list of the cmdlets is good, but we also need a list of aliases that we can use. We do not want to type the entire `Set-FileTime` command when a simple touch command will do just fine, right? We can get the aliases using the `Get-PscxAlias` function:

```
PS> Get-PscxAlias
```

CommandType	Name	Definition
-----	----	-----
Alias	?:	Invoke-Ternary
Alias	??	Invoke-NullCoalescing
Alias	adl	Add-DirectoryLength
Alias	apropos	Get-HelpMatch
Alias	apv	Add-PathVariable
...		

Remember the `$PscxTextEditorPreference` variable you saw previously? There are many cmdlets that rely on similar preference variables. In addition, PSCX defines several variables that contain information about the environment. To get all the special variables, we need to call the `Get-PscxVariable` function:

```
PS> Get-PscxVariable
```

Name	Value
----	-----
__PscxCdBackwardStack	{C:\}
__PscxCdForwardStack	{}
__PscxDebugBreakpointsEnabled	True
__PscxDebugSkipCount	0
__PscxProfileRanOnce	
IsAdmin	False
NTAccount	NULL\Hristo
NTIdentity	System.Security.Principal.WindowsIde...
NTPrincipal	System.Security.Principal.WindowsPri...
ProfileDir	C:\Users\Hristo\Documents\WindowsPow...
...	

The variables contain a lot of useful information, and it is best if you go through the list and try to spot the ones that can be of use. Take the `$IsAdmin` variable for example. We had to do some crazy .NET programming in Chapter 11 to see if the current user is a computer administrator. We do not need to do that once we have PSCX installed. There is one important caveat here: do not touch variables starting with double underscores. Those variables are used internally by the PSCX cmdlets and functions.

The last of the information functions is `Get-PscxDrive`. It will list the drives pointing to locations that are served from item providers implemented by PSCX. Here are the default drives that PSCX registers for the global assembly cache and Internet Explorer's feed store:

```
PS> Get-PscxDrive
```

Name	Provider	Root	CurrentLocation
----	-----	----	-----
Feed	FeedStore		
Gac	AssemblyCache	Gac	

We will get to describing each of the PSCX providers later in this chapter. For further details on item providers, please refer to Chapter 7.

Maybe the most powerful tool in learning both PSCX and PowerShell is the `Get-HelpMatch` function. Aliased to `apropos`, it will search for occurrences of a string or a regular expression in all help topics. Figure 21-1 shows it in action, looking for help on `New-Object`. Help searches are usually slow, and the function shows a nice progress bar as it traverses the help topics.

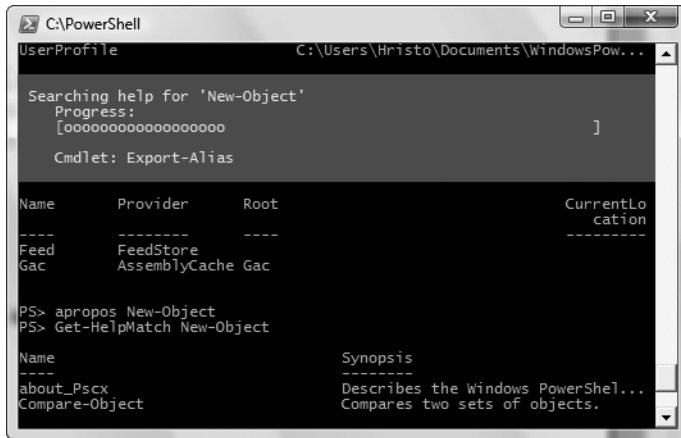


Figure 21-1. *Get-HelpMatch* searching for help on `New-Object`

File System Utilities

Working with files and folders usually occupies most of our time in the shell. Naturally, PSCX provides several enhancements that make that experience more enjoyable.

Navigation Helpers

The navigation support in PowerShell is somewhat poor when compared to the support found in UNIX shells. I, as a user, have long been annoyed by the lack of support for the `cd -` command, the fastest way to return to the previous folder you were visiting. People used to `cmd.exe`'s handling of the `cd` command may be used to `cd. .` (no space between `cd` and `. .`) and may be ticked by the fact that it does not work in PowerShell. PSCX solves these problems by creating its own `cd` function that replaces the original version. The new function keeps track of the folders you visit and provides some other interesting features.

You get a history of the folders you have visited, and you can navigate both back and forward in that history using the `-` and `+` symbols as shortcuts. Here is an example:

```
PS> cd \users\hristo
C:\users\Hristo
PS> cd ..
C:\Users
PS> cd -
C:\users\Hristo
PS> cd +
C:\Users
```

You can also safely omit the space after the `cd` command:

```
PS> cd\
C:\
PS> cd users\hristo
C:\users\Hristo
PS> cd..
C:\Users
PS> cd-
C:\users\Hristo
PS> cd+
C:\Users
```

Note that there are no spaces in the `cd\`, `cd-`, `cd+`, and `cd..` commands.

NTFS Helpers

Ever since the release of Windows 2000, the NTFS file system has supported two special types of file system entries: hard links and directory junctions. Hard links represent a way to associate more than one file system entry with the same block of data on disk. The entries look like different files and modifying the file contents through one of the entries affects all of them. Deleting an entry deletes the file data if and only if the deleted entry is the last one that points to the data. If there are more entries, the data will remain intact and will still be accessible through them. PSCX provides support for creating hard links through the `New-HardLink` cmdlet. Here is how to use it:

```
PS> New-Hardlink test-link.txt test.txt
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-ar--	7/7/2007 11:33 PM	4	test-link.txt

```
PS> dir
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::Z:\22 - PowerShell
1 Community Extensions\TestFolder
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-ar--	7/7/2007 11:33 PM	4	test-link.txt
-ar--	7/7/2007 11:33 PM	4	test.txt

The `test-link.txt` directory entry points to the data in `test.txt`. To get rid of the entry, just delete it. Do not worry; the original file will not be affected.

One big drawback of hard links is that you cannot tell if a file is the original directory entry, so most file systems, NTFS included, do not allow hard-linking directories; there is no way to detect a circular reference that would cause an infinite loop. That is why NTFS provides junction

points or directory junctions. Junctions are similar to hard links but work for directories only. They are marked in a special way, so we always know which one is the original. We can create a junction using the `New-Junction` cmdlet:

```
PS> mkdir folder1
...
PS> New-Junction folder2 folder1
```

Mode	LastWriteTime	Length	Name
d----	12/17/2007 10:15 PM	<JUNCTION>	folder2 [Z:\21 - PowerShell Community Extensions\folder1]

The cmdlet returns the newly created junction. It is not safe to delete a junction, because some programs delete folders recursively, and that may cause the deletion of the original files instead of the junction itself. To get rid of our junction, we have to use the `Remove-RepasePoint` cmdlet:

```
PS> Remove-RepasePoint folder2
PS> dir
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::Z:\22 - PowerShell
l Community Extensions\TestFolder
```

Mode	LastWriteTime	Length	Name
d----	11/7/2007 2:26 AM	<DIR>	folder1

Something is fishy here; we are creating a junction and deleting a repase point! There is no need to have a dedicated `Remove-Junction` cmdlet, as directory junctions are just a type of repase points, and we do not need additional functionality to delete them.

Note Repase points offer a convenient way to expose various functionality with a familiar file and folder I/O interface. A repase point is a special file system object that associates a file system entry with some code that performs different operations on the stream of data while pretending to be reading or writing it to disk. In the case of a junction, the repase point contains the directory redirection code. In short, a junction appears as a normal folder, but it redirects all reads and writes to the real folder when accessed.

Windows Vista introduces symlink support to the NTFS file system. Symlinks or symbolic links do not have the limitations of hard links and junctions and can link to both files and folders. They can even link to locations across the network. We can create a symlink using the `New-Symlink` cmdlet. Please note that creating a symlink requires privilege elevation. Here is an example that again links `folder2` to `folder1`:

```
PS> New-Symlink folder2 folder1
```

Mode	LastWriteTime	Length	Name
d----	11/7/2007 2:34 AM	<SYMLINKD>	folder2 [C:\PowerShell\symlink\folder1]

Here is how we delete a symlink:

```
PS> Remove-RepasePoint folder2
PS>
```

Note that we need to call `Remove-RepasePoint` again. I believe you know why already—symlinks are implemented with repase points too.

Compressing and Archiving Files

Surprisingly, PowerShell does not include any support for creating or reading from compressed files and archives. I can imagine some nice providers that will allow you to mount an archive as a separate drive and manipulate files as if they were not compressed at all, but *c'est la vie*. Luckily, the PSCX guys are here to save the day.

The most important cmdlet that we need to learn here is `Write-Zip`. At minimum, it takes a source folder and a destination path. Here is an example:

```
PS> Write-Zip C:\PowerShell PowerShell.zip
```

Mode	LastWriteTime	Length	Name
-a---	11/7/2007 2:45 AM	1089671	PowerShell.zip

The cmdlet puts up a nice progress bar while it is doing its job, as shown in Figure 21-2.

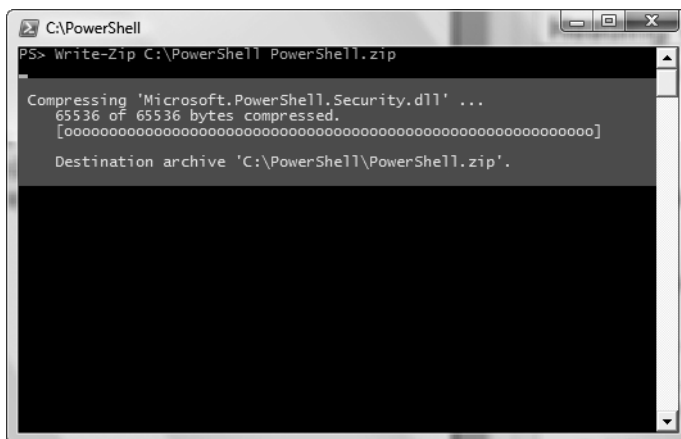


Figure 21-2. Creating a ZIP archive using the `Write-Zip` cmdlet

The ZIP archive format is not the only option that we have for archives. The other supported format is Tar. People with UNIX experience will recognize it, since it is the prevalent format in

that world. We can create a Tar archive using the Write-Tar cmdlet. We can call it in absolutely the same way as we did with Write-Zip. Here is the code that archives our folder in a Tar file.

```
PS> Write-Tar C:\PowerShell PowerShell.tar
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	11/7/2007 2:51 AM	4321280	PowerShell.tar

You might notice that the Tar archive is significantly larger in size than the ZIP one. That is so because Tar archives are not compressed. It is a custom in the UNIX world to separate the archive format from the compression algorithm. PSCX offers two compression-only cmdlets: Write-GZip and Write-BZip2. They can compress one or many files using the two popular compression algorithms gzip and bzip2, respectively. This is how we can compress our archive in the two formats:

```
PS> Write-GZip PowerShell.tar PowerShell.tar.gz
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	11/7/2007 2:54 AM	2159677	PowerShell.tar.gz

```
PS> Write-BZip2 PowerShell.tar PowerShell.tar.bz2
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	11/7/2007 2:55 AM	2180229	PowerShell.tar.bz2

Let me stress again that the gzip and bzip2 algorithms cannot archive many files. We need to create a Tar archive first and then put it up for compression.

Note Unfortunately, not everything is perfect on the compression front. We lack the decompression part of the puzzle. There is no way to look inside or decompress our archive using PowerShell. For the time being, we will have to resort to using external programs.

Network Utilities

The network part of our automation tasks has not been forgotten by the PSCX developers. We have pretty good support for the Windows network, especially when it comes to working with Active Directory. If we are in a Windows network domain, we can ask the Active Directory for our DHCP server or the domain controller. We can do that with the Get-DhcpServer and Get-DomainController cmdlets, respectively. Here is how we can get that information:

```
PS> Get-DhcpServer
```

ServerName	Address
-----	-----
win2k3-test.TEST.local	192.168.0.1

```
PS> Get-DomainController
```

```

ServerName      : WIN2K3-TEST
DnsHostName     : win2k3-test.TEST.local
Site            : Default-First-Site
Domain          : TEST.local
DN              : CN=WIN2K3-TEST,OU=Domain Controllers,DC=TEST,DC=local
GlobalCatalog   : True

```

Note that the `Get-DhcpServer` and `Get-DomainController` cmdlets will fail if your machine is not a member of a Windows domain.

PSCX also offers some small but handy network-related utilities that we can use to ping machines on the network or resolve DNS names. We can use the built-in Windows ping utility, but we would have to parse the results using regular expression. Why do that if we can have our own object-oriented ping? Here is how to use the ping utility and query its results:

```

PS> $pingResults = ping google.com
Pinging google.com with 32 bytes of data:
    Reply from 72.14.207.99 bytes=32 time=149ms TTL=238
    Reply from 72.14.207.99 bytes=32 time=141ms TTL=238
    Reply from 72.14.207.99 bytes=32 time=142ms TTL=238
    Reply from 72.14.207.99 bytes=32 time=144ms TTL=238

PS> $pingResults.MinimumTime
141
PS> $pingResults.MaximumTime
149
PS> $pingResults.Sent
4
PS> $pingResults.Received
4
PS> $pingResults.Loss
0

```

The ping command you see in the preceding code is not the Windows ping tool. This is an alias for the `PSCXPing-Host` cmdlet. It returns a `PingHostStatistics` object that holds the result of our query.

We can do a similar thing when resolving DNS names to IP addresses by using the `Resolve-Host` cmdlet. Here is how to get some of the Google IP addresses:

```
PS> $result = Resolve-Host google.com
PS> $result.AddressList
```

```
IPAddressToString : 64.233.187.99
Address           : 1673259328
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False
```

```
IPAddressToString : 72.14.207.99
Address           : 1674513992
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False
```

```
IPAddressToString : 64.233.167.99
Address           : 1671948608
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False
```

PSCX also ships a script that can download web page content from a HTTP server—`Get-Url`. It is very easy to use; it needs only a URL. Here is how to get the contents of the Yahoo! start page:

```
PS> Get-Url http://www.yahoo.com
<html>
<head>
<title>Yahoo!</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

I would still recommend using the `Get-Url.ps1` script that we created in Chapter 18. In addition to getting a string representation, our script can save the contents to a file, provide custom HTTP header values, and work with proxy servers and HTTP authentication.

Executing Processes and Commands

You might remember our little problem from Chapter 14: with all the built-in process-related cmdlets that PowerShell has, getting a reference to the process you just started is still quite hard. We had two solutions: look for the process using its expected window title or process name shortly after you start it or use a piece of .NET code that starts the process like this:

```
PS C:\> $notepad = [Diagnostics.Process]::Start("notepad.exe")
PS C:\> $notepad
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
51	3	1860	7188	42	0.13	2816	notepad

The preceding solution works fine with the minor inconvenience that it is hard to pass command-line parameters to the program you want to start. `PSCX` solves that exact problem by offering a new cmdlet, `Start-Process`. The cmdlet returns the process object it just created. It does much more too. For example, it can spawn a process under different user credentials:

```
PS> $credentials = Get-Credential

cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
PS> $notepad = Start-Process -Path notepad.exe -Credential $credentials
```

The new process spawned by the code will run under the user credentials you supplied in the user name/password dialog box that `Get-Credential` pops open for the user. Here is how we can pass a command-line parameter using the `Arguments` parameter:

```
PS> $notepad = Start-Process -path notepad -arguments "C:\PowerShell\test.txt"
PS> $notepad | Stop-Process
PS> $notepad.HasExited
True
```

The preceding code opens the `C:\PowerShell\test.txt` file in Notepad and then kills the process. After that, it confirms the process has really terminated by checking the `HasExited` property.

Another nice feature that the cmdlet offers is being able to start a new process using a specific priority. We can do that by setting the `Priority` parameter to something like `Idle`, `BelowNormal`, `Normal`, `AboveNormal`, `High`, or `RealTime`. Here is a snippet that starts two Notepad processes with different priorities:

```
PS> $notepad1 = Start-Process notepad
PS> $notepad2 = Start-Process notepad -Priority BelowNormal
PS> $notepad1.BasePriority
8
PS> $notepad2.BasePriority
6
PS>
```

We check the `BasePriority` for both processes; 6 is the code for `BelowNormal` process priority and 8 for `Normal`.

The last interesting thing I should mention is that we can start a windowed process with its main window being hidden, so that the process runs in the background. This can be useful if

you want to prevent a process from stealing focus and distracting the user. Of course, you have to be careful with that, as the process may get hung in the background and there will be no way to know that. Here is the code that will start a hidden Notepad process:

```
PS> $notepad = Start-Process Notepad -WindowStyle Hidden
PS> $notepad.CloseMainWindow()
False
PS> $notepad.HasExited
False
PS> $notepad | Stop-Process
PS> $notepad.HasExited
True
```

The preceding code does not display anything, but it really creates a new process. The new process is visible in the Windows Task Manager, and we can also get it using the `Get-Process` cmdlet. Also note that the code that tries to close the main window, as a graceful way to stop the process, does not work when the main window is initially hidden, and the process keeps running. That leaves killing the process as the only option to exit it.

Let's get back to starting processes with different credentials. You might agree that even though `Start-Process` makes doing so easier, it is still not good enough. We need a quicker way to run a program or a shell command with elevated privileges. `PSCX` solves that by providing an `elevate` function that wraps a call to `Start-Process` with a friendlier interface. It accepts a shell command or a program name. In the former case, it spawns a new shell with elevated privileges to execute the command. In the latter one, it just starts the program with elevated privileges. Remember that I mentioned that only administrators with elevated privileges could create symlinks on Windows Vista? First, let's see what happens when we try that when running under normal privileges:

```
PS> New-Symlink d users
New-Symlink : A required privilege is not held by the client. (Excepti
on from HRESULT: 0x80070522)
At line:1 char:12
+ New-Symlink <<<< d users
```

The error message says it all—no permissions! Let's now use `elevate` to fix that:

```
PS> elevate New-Symlink d users
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
0	0	224	60	2	0.00	3064	powershell

The function returns the `Process` object for the newly spawned shell, so that we can kill it if the command takes too long. The new shell now successfully executes the command:

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	11/7/2007 8:53 PM	<SYMLINKD>	d [C:\users]

XML Tools

PowerShell offers really great XML support through its type adapters, but there is always room for improvement. One such example is pretty printing. Quite often, we get XML that is so horribly formatted that it is close to being unreadable. For example, here is how a snippet of the XHTML (HTML expressed as valid XML) code taken from the W3C looks:

```
PS> type unformatted.xml
<div class="navBlock">
<h2 class="spot-head">W3C Technical Plenary</h2>
<div class="spot-image">
<a href="http://www.bea.com/">
</a>
</div>

<div class="spot-image">
<a href="http://www.cisco.com/">
</a>
</div>
</div>
```

We can clean this up using the `PSCX Format-Xml` cmdlet. It works by being given a path to the XML file and it outputs it to the console. Here is how we can use it to format the preceding snippet:

```
PS> Format-Xml unformatted.xml
<div class="navBlock">
  <h2 class="spot-head">W3C Technical Plenary</h2>
  <div class="spot-image">
    <a href="http://www.bea.com/">
      
    </a>
  </div>
  <div class="spot-image">
    <a href="http://www.cisco.com/">
      
    </a>
  </div>
</div>
```

By default, the cmdlet uses two spaces per indentation level. You can override that by providing the `-Indent` parameter; it accepts a string that will be used instead of the two spaces. If you want to use tabs, pass the ``t` string:

```

PS> Format-Xml unformatted.xml -Indent `t > formatted.xml
PS> type formatted.xml
<div class="navBlock">
    <h2 class="spot-head">W3C Technical Plenary</h2>
    <div class="spot-image">
        <a href="http://www.bea.com/">
            <img width="120" height="75" alt="BEA logo" src
="/2007/11/TPAC/bea.
png" />
        </a>
    </div>
    <div class="spot-image">
        <a href="http://www.cisco.com/">
            <img width="120" height="63" alt="Cisco logo" s
rc="/2007/11/Cisco-l
ogo.png" />
        </a>
    </div>
</div>

```

The preceding example uses the standard text redirection mechanism to save the formatted XML to another file.

The built-in type adapters are real time savers in most situations, but sometimes, they go too far. There are occasions where we can use XML-based technologies as an easier way to perform a task. Such examples are XML validation, extracting data using XPath, or transforming a document using XSLT. There is no need to shy away from technologies like DTD and XML Schema for validation, XPath for selecting elements, and XSLT for transforming documents into something else, especially if you already have experience with those technologies. Starting with validation, PSCX offers a `Test-Xml` cmdlet. At minimum, it can validate an XML file to see if it's well formed. Here is how to do that for a small document:

```

PS> type message.xml
<?xml version="1.0"?>
<note
    xmlns="http://www.test.com"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>
    <to>Mike</to>
    <from>Jim</from>
    <heading>Reminder</heading>
    <body>Meeting @ 6 PM tomorrow!</body>
</note>
PS> Test-Xml message.xml
True

```

As you can see, the cmdlet returns `$true` if everything is fine or `$false` otherwise. We can now use an XML schema to validate the message file. Here is the sample schema:

```

PS> type message.xsd
<?xml version="1.0"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.test.com"
  xmlns="http://www.test.com"
  elementFormDefault="qualified">
  <xs:element name="note">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="to" type="xs:string"/>
        <xs:element name="from" type="xs:string"/>
        <xs:element name="heading" type="xs:string"/>
        <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

The schema requires the document to have four elements containing strings: to, from, heading, and body. Let's now add an extra element that should not be there and see how we can validate the document using the schema:

```

PS> type message.xml
<?xml version="1.0"?>
<note
  xmlns="http://www.test.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  >
  <to>Mike</to>
  <from>Jim</from>
  <heading>Reminder</heading>
  <body>Meeting @ 6 PM tomorrow!</body>
  <illegal id="1">element should not be here</illegal>
  <illegal id="2">element2 should not be here</illegal>
</note>
PS> Test-Xml message.xml -SchemaPath message.xsd -Verbose
VERBOSE: Test-Xml processing path 'Z:\22 - PowerShell Community
Extensions\message.xml'
VERBOSE: Error: The element 'note' in namespace 'http://www.test.com'
has invalid child element 'illegal' in namespace
'http://www.test.com'.
False

```

We passed the schema in the `-SchemaPath` parameter. Additionally, we used the `-Verbose` parameter to get the exact error message. If we had not done that, we would have gotten only the `$false` result.

Navigating and modifying an XML document using the property-based syntax is pretty quick, but sometimes we may need the full power of XPath to select a node according to some criteria. Sure, we can loop over nodes and traverse them recursively until we find the right one, but quite often, XPath allows us to write a small query that will do the job nicely. PSCX makes that easy by providing a `Select-Xml` cmdlet. Let's use our previous example to illustrate how to use XPath. The example is an extra special case, because it uses XML namespaces—our note element contents are in the namespace with the URI of `http://www.test.com`. To make XPath work, we need to define an XML namespace to tag prefix mapping and use that in our XPath expression. Here is how we can do that to select the `<illegal>` element that has an `id` attribute of 2:

```
PS> Select-Xml -Path message.xml -XPath "//test:illegal[@id=2]" `
-Namespace "test=http://www.test.com"
```

Name	NodeType	OuterXml
-----	-----	-----
illegal	Element	<illegal id="2" xmlns="http://www.test.com">element2 should not be here</illegal>

The `Namespace` parameter sets up the tag prefix to namespace mapping. We are using the test prefix. The XPath expression now becomes `//test:illegal[@id=2]`.

Note Without pretending in any way to teach XPath here, I want to quickly explain that the `//` part means that we want to match an `<illegal>` element at an arbitrary depth in the document. The `[@id=2]` expression means that we are interested in elements that have an `id` attribute that equals 2. To get a quick and easy tutorial on XPath, you can visit the one on the W3Schools site at <http://www.w3schools.com/xpath/>.

Selecting an element without XPath would have to involve looping and manually checking attribute values:

```
PS> $doc = [xml] (Get-Content message.xml)
PS> $doc.note.illegal | where { $_.id -eq 2 }
```

id	#text
--	----
2	element2 should not be here

This code is more complex and less powerful than the previous XPath expression. The `where` part assumes that all `<illegal>` elements will be direct children of the `<note>` element. The XPath version would have returned all elements, even if they were nested deeper.

Finally, we are often tasked with taking an XML document and converting it into something else—another XML document (in a different format), a text report, or a richer HTML-based representation of the data. We can loop through the document elements and generate the resulting document, but again, that is not a very effective or powerful solution. PSCX offers a `Convert-Xml` cmdlet that allows us to use an XSLT transformation. XSLT transformations are

a declarative way to describe a set of templates that will be applied to a set of XML elements to generate another XML document.

Note Many books have been written about XSLT, and teaching it is outside the scope of this book. For now, just have in mind that XSLT is a really effective way to manipulate XML, and learning it may be worth the time. For a good tutorial and an excellent starting point for learning XSLT, you can refer to the W3Schools site again; the XSLT tutorial is available at <http://www.w3schools.com/xsl/>.

To show our cmdlet in action, suppose we are trying to generate a red paragraph for every `<illegal>` element that we have in our message XML document. Here is our XSLT transformation file:

```
PS> type message.xslt
<?xml version="1.0"?>

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:test="http://www.test.com">
<xsl:template match="/">
  <html>
    <head>
      <title>Illegal elements</title>
    </head>
    <body>
      <xsl:for-each select="//test:illegal">
        <p style="color: red">
          <xsl:value-of select="."/>
        </p>
      </xsl:for-each>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

The real action happens in the `<xsl:for-each>` element—it selects all `<illegal>` elements in the documents and generates a paragraph element with a red font color. Again, note that we have to handle XML namespaces in a special way: we declare the test namespace prefix with the `xmlns:test="http://www.test.com"` attribute and then use the prefix in the XPath expression. Finally, let's see the transform in action:

```
PS> Convert-Xml -path message.xml -XsltPath message.xslt
<html xmlns:test="http://www.test.com">
  <head>
    <title>Illegal elements</title>
  </head>
```

```
<body>
  <p style="color: red">element should not be here</p>
  <p style="color: red">element2 should not be here</p>
</body>
</html>
```

Working with Image Files

With the advent of digital photography, everyone has become quite skilled at working with image files. Everyone seems to have a vast collection of photos, and organizing those can easily get out of hand. We need tools to validate that an image's format, resolution, or dimensions comply with our criteria. We can easily do that with one of the freely available desktop programs; they all offer a nice graphical user interface that shows us how to verify image files in an intuitive way. Here, we hit the problem—those nice interfaces are not scriptable at all. We cannot use them for collections of tens or even hundreds of images. We can use external console programs that have been specifically designed with scriptability in mind to get the job done here. The first that comes to mind is ImageMagick: it is the Swiss Army knife of image manipulation utilities and can resize, resample, or convert images between formats, as well as add all sorts of effects like text, titles, rounded corners, drop shadows, and much more. PSCX recognizes the need for such a tool that is deeply integrated with .NET and PowerShell and includes several cmdlets that can help us with our basic image manipulation needs. At the time of this writing, we can read information about images, convert images from one format to another, and resize images.

The first thing that we need is the ability to read images from files into .NET objects. We can do this with the `Import-Bitmap` cmdlet. It accepts a file name or a wildcard and returns one or many `System.Drawing.Bitmap` objects. The cmdlet honors the tradition of using standard .NET objects where available. Let's now see how we can get some information about a screenshot we have in the current folder:

```
PS> Import-Bitmap screenshot.tif
```

```
Tag :
PhysicalDimension : {Width=1119, Height=828}
Size : {Width=1119, Height=828}
Width : 1119
Height : 828
HorizontalResolution : 96
VerticalResolution : 96
Flags : 77840
RawFormat : [ImageFormat: b96b3cb1-0728-11d3-9d7b-0000f81ef32e]
PixelFormat : Format24bppRgb
Palette : System.Drawing.Imaging.ColorPalette
FrameDimensionsList : {7462dc86-6180-4c7e-8e3f-ee7333a7a483}
PropertyIdList : {254, 256, 257, 258...}
PropertyItems : {254, 256, 257, 258...}
```

You can see the image has a horizontal and vertical resolution of 96 pixels per inch, a width of 1119 pixels, and height of 828 pixels. The pixel format uses 24 bits per pixel: 8 bits for each of the red, green, and blue color components. The `PropertyIdList` and `PropertyItems` properties look interesting; they hold the exchangeable image file format (EXIF) metadata, but, unfortunately, .NET and PowerShell do not offer good support for reading that data yet.

Now that we are able to read various pieces of data from images, we can use the full power of the script language and its pipelines to craft some serious commands. For example, I have to use a special program to capture screenshots for this book. Those screenshots must have a 150 pixels-per-inch resolution and be saved in the TIFF image format. I used to have a problem overcoming my existing habits of taking screenshots using the Alt+Print Screen shortcut key, which produces files having a 96 pixels-per-inch resolution. Until I was able to eradicate my habit, I had to run checks for all TIFF files that told me if I had a bad screenshot somewhere. Here is how I used PowerShell to help me through this:

```
PS> $tifs = dir *.tif
PS> $tifs | where { (Import-Bitmap $_).HorizontalResolution -ne 150 }
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::Z:\22 - PowerShell
1 Community Extensions
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	11/11/2007 6:47 PM	173844	bad_screenshot.tif
-a---	11/11/2007 6:47 PM	173844	screenshot.tif

The command uses the `where` cmdlet to filter the list of all TIFF images in the current folder and return only those that have a horizontal resolution different than 150 pixels-per-inch. Theoretically, I should be using both the `HorizontalResolution` and `VerticalResolution` properties, but I have yet to see an image file with different horizontal and vertical resolutions in practice.

The `Import-Bitmap` cmdlet returns fully functional `Bitmap` objects that hold a reference to the binary data for the corresponding image. That means we can use the power of .NET to modify the image and save it in a different format. Luckily, we do not need to resort to working with raw .NET objects here. `PSCX` has two useful cmdlets that allow us to resize an image and then save it in a specified format—the `Resize-Bitmap` and `Export-Bitmap` cmdlets. The former accepts a `Bitmap` object, modifies it, and returns another `Bitmap` object. The latter just takes a `bitmap` and saves it in the specified format. We can use these to take care of one of the most common image-related operations—creating a thumbnail for an image. Here is the code that does this:

```
PS> $original = Import-Bitmap screenshot.tif
PS> $thumb = Resize-Bitmap $original -Percent 10
PS> Export-Bitmap $thumb -Path thumbnail.png -Format Png
PS> $original | Select Width,Height
```


Width	Height
-----	-----
1119	828

```
PS> $thumb | Select Width,Height
```

Width	Height
-----	-----
111	82

Look at the Width and Height properties for the original image and the final result. The code generates a thumbnail with a size that is ten percent of the original image dimensions. Images encoded in the TIFF format are usually large files; that format is used when you need precision and no loss of quality. Hoping that we would get a smaller thumbnail file by switching to another format, we save the image in the PNG format.

Clipboard Helpers

I do not remember anymore how many times I have wanted to have the output of a command placed in the Windows clipboard so that I could paste it in some other program. Until recently, that was downright impossible without using a third-party program that accepts piped text and sets that on the clipboard buffer. Windows Vista finally fixes that and ships with a nice utility called `clip.exe`. It does just what we need most of the time; it sets all the text it has been piped in as the clipboard contents. Unfortunately, we cannot say we have this problem solved. First, `clip.exe` does not handle anything but text, and the Windows clipboard can contain various file formats: rich text, HTML, images, files, and many more. Second, at the time of this writing, Windows Vista is yet to see wide adoption, and we may need to maintain many Windows XP machines. PSCX steps in with its clipboard helper cmdlets and allows us to do our job on different versions of Windows without installing any other programs.

The most basic need that we have is getting clipboard contents. We can do that with the `Get-Clipboard` cmdlet. Whenever you copy a piece of text, like the current paragraph for example, you can get it from your script by calling the cmdlet. Here is how:

```
PS> Get-Clipboard
```

The most basic need that we have is getting clipboard contents. We can do that with the `Get-Clipboard` cmdlet. Whenever you copy a piece of text, like the current paragraph for example, you can get it from your script by calling the cmdlet. Here is how:

Note that many programs will not keep plain text on the clipboard even though they may seem to contain text. Microsoft Word, for example, will put copied text as HTML and RTF text. The default `Get-Clipboard` cmdlet behavior will return a fully formatted HTML document that contains a lot of formatting that you do not need along with some custom HTML tags that only Microsoft products understand. The `Get-Clipboard` authors have provided a nice switch, `FragmentOnly`, that allows us to get only the copied text. Let's now copy the first two sentences of this paragraph and get the fragment as an HTML string:

```
PS> Get-Clipboard -Html -FragmentOnly
<span lang=EN-US style='font-size:12.0pt;mso-bidi-font-size:
10.0pt;font-family:"Times New Roman";mso-fareast-font-family:"Times Ne
w Roman";
mso-ansi-language:EN-US;mso-fareast-language:EN-US;mso-bidi-language:A
R-SA'>Note
that many programs will not keep plain text on the clipboard even thou
gh they
may seem to contain text. Microsoft Word for example will put copied t
ext as
HTML and RTF text.</span>
```

Apart from the somewhat long style attribute in there, the HTML code looks very clean.

We are not limited to text-based formats with Get-Clipboard. It can return various objects depending on what it finds on the clipboard. The ones we can make the most use of are images and files. We can get an image that has been copied on the clipboard as a System.Drawing.Bitmap object. For example, press the Print Screen key; it will take a screenshot of your desktop and place an image on the clipboard. You can get that and save it to a file using this command:

```
PS> $screen = Get-Clipboard -Image
PS> $screen | select Width,Height,PixelFormat
```

Width	Height	PixelFormat
-----	-----	-----
1440	900	Format32bppRgb

```
PS> Export-Bitmap -Bitmap $screen -Path screenshot.png -Format Png
```

In my case, I got a 1440 × 900 image in 32 bits-per-pixel color depth. Note that we have to provide the Image parameter to Get-Clipboard; it will default to text otherwise and return a \$null value. At the end, we saved the image using another PSCX cmdlet, Export-Bitmap.

Getting file objects from the clipboard is easy too. We get them as regular FileInfo objects that we can pass to other PowerShell cmdlets. This can be a useful technique for providing parameters to scripts. For example, we can copy several files from Windows Explorer and invoke commands that will print various statistics. I just copied two image files, and here is how I can get information about them:

```
PS> Get-Clipboard -Files
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	11/11/2007 6:47 PM	173844	screenshot.tif
-a---	11/12/2007 5:56 AM	184792	screenshot.png

```
PS> Get-Clipboard -Files | Import-Bitmap | select Size
```

```
Size
----
{Width=1119, Height=828}
{Width=1440, Height=900}
```

As you can see, piping the file collection to other cmdlets allows us to get detailed information. An important thing to note here is that, just as the case with image data, we need to pass the `Files` parameter to `Get-Clipboard`, or we will not get anything back.

Not only can we get the clipboard contents in various formats but we can put values in there too. This is where the `Set-Clipboard` cmdlet comes into play. The most common case is to collect text from a shell command and save it to the clipboard. Here is how we can do that:

```
PS> "Set-Clipboard text" | Set-Clipboard
PS> Set-Clipboard -Text "Set-Clipboard text"
```

Both lines do the same thing; they clear the existing clipboard contents and replace them with the string we piped in or provided as the `Text` parameter. We can also set HTML content:

```
PS> Set-Clipboard -Html "<i>Set-Clipboard</i> rocks"
```

The `<i>` tag marks text as italics. If you are pasting the text in programs like Microsoft Word that can handle HTML data, the cmdlet name will be italicized.

The real fun with `Set-Clipboard` comes when we start to work with images and files. Here is how we can get an image and place it on the clipboard:

```
PS> $image = Import-Bitmap screenshot.tif
PS> Set-Clipboard -Image $image
PS>
```

Note that we are using another PSCX cmdlet, `Import-Bitmap`, to load the image in the first place. Now that we have the image available in our clipboard, we can paste it in programs that support image data: Paint and Microsoft Word are the first that come to mind.

Placing files on the clipboard may sound odd the first time you encounter it, but it is a useful technique to keep in mind. We can feed a collection of files that we have obtained from another cmdlet to `Set-Clipboard` and make those files available to other programs. Many programs accept file pastes, and that means we can feed them data originating from PowerShell. For example, here is how we can get all PNG images in the current folder and put them on the clipboard:

```
PS> Set-Clipboard -Files (dir *.png)
PS>
```

Pasting the files in a program will trigger different actions depending on the program itself. Pasting those files in Windows Explorer will copy them to the current folder just like a regular file copy and paste. Pasting them in Microsoft Word will embed the files in the document.

PSCX offers two helper cmdlets that set the clipboard contents when being passed various objects: `Out-Clipboard` and `Write-Clipboard`. The former will convert objects to text using the `Out-String` cmdlet effectively formatting them just like they would look on the console before placing the text on the clipboard. The latter will use the built-in string representation of an object, bypassing PowerShell's formatting. Here is how we can get the five processes using the most memory and place a table containing the data on the clipboard using PowerShell's formatting:

```
PS> Get-Process | sort WS -Desc | select -First 5 | Out-Clipboard
PS>
```

The clipboard contains text like this:

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
467	21	56752	79056	390	206.55	2076	WINWORD
155	8	103696	60644	208	158.73	1968	dwm
511	14	70368	58696	177		1004	svchost
791	36	46912	53656	279	80.55	2008	explorer
760	11	66444	53304	242	37.16	3052	powershell

To contrast this output with the default string representation, let's perform the same operation, but this time, we'll pipe the objects through Write-Clipboard:

```
PS> Get-Process | sort WS -Desc | select -First 5 | Write-Clipboard
PS>
```

The clipboard now contains text that looks like this:

```
System.Diagnostics.Process (WINWORD)
System.Diagnostics.Process (dwm)
System.Diagnostics.Process (svchost)
System.Diagnostics.Process (explorer)
System.Diagnostics.Process (powershell)
```

The Write-Clipboard cmdlet obtains the text representation by calling the `Object.ToString()` method for all objects.

New Providers

The PowerShell Community Extensions project contains quite a few of cmdlets, functions, and scripts. That is not all—the snap-in assembly contains three providers that can help us out a lot. Two of the providers allow us to manipulate machine-specific information stores: one works with the .NET global assembly cache, the other can access the Internet Explorer 7 feed store. The third provider works in a Windows domain only, and it allows us to work with the domain's Active Directory in a way that is familiar to us. The providers make those stores look like file systems inside PowerShell, and we can use familiar cmdlets to navigate, extract data, and manipulate items.

Reading from the .NET Global Assembly Cache

The AssemblyCache provider allows us to access the .NET global assembly cache. It automatically registers a new drive, `Gac`. The assemblies are exposed as `AssemblyName` objects that we can use to obtain various information. Here is how to get all the PowerShell assemblies:

```
PS> cd Gac:
```

```
PS> dir *PowerShell*
```

Version	PublicKeyToken	Arch	Culture	Name
-----	-----	----	-----	----
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...
1.0.0.0	31bf3856ad364e35	MSIL		Microsoft.PowerShell...

We can run different queries by using cmdlets like `select` or `foreach`. Here is how to get just the names for the PowerShell assemblies:

```
PS> dir *PowerShell* | select Name
```

```
Name
----
Microsoft.PowerShell.Security
Microsoft.PowerShell.Commands.Utility.resources
Microsoft.PowerShell.Commands.Utility.resources
Microsoft.PowerShell.ConsoleHost
Microsoft.PowerShell.Security.resources
Microsoft.PowerShell.Security.resources
Microsoft.PowerShell.Commands.Utility
Microsoft.PowerShell.Commands.Management.resources
Microsoft.PowerShell.Commands.Management.resources
Microsoft.PowerShell.ConsoleHost.resources
Microsoft.PowerShell.ConsoleHost.resources
Microsoft.PowerShell.Commands.Management
```

Access to the global assembly cache is read only for the time being; there is no way to add, remove, or modify an assembly there.

Exploiting the Internet Explorer Feed Store

Internet Explorer 7 comes with a global feed store. The concept is simple: RSS feed subscription data and item caches are stored in a central location so that they can be shared by many applications. On initialization, the PSCX snap-in registers a new Feed drive that we can use to access the data. Here is how that looks:

```
PS> cd Feed:
```

```
PS> dir
```

Type	Name	ItemCount	UnreadItemCount
folder	Microsoft Feeds	95	89

```
PS> cd 'Microsoft Feeds'
```

```
Microsoft Feeds
```

```
PS> dir
```

Type	Name	ItemCount	UnreadItemCount
feed	Microsoft at Home	45	43
feed	Microsoft at Work	50	46

Feeds that we subscribe to look like folders, and feed items like, well, items. Unlike the Gac drive, we can manipulate items in various ways. For example, we can use the standard `New-Item` cmdlet to subscribe to a new feed. Here is how to add a subscription for the PowerShell team blog's Atom feed:

```
PS> $url = "http://blogs.msdn.com/powershell/atom.xml"
```

```
PS> pwd
```

```
Path
```

```
----
```

```
Feed:\Microsoft Feeds
```

```
PS> New-Item -itemType feed -path "PowerShell Blog" -value $url
```

Type	Name	ItemCount	UnreadItemCount
feed	PowerShell Blog	0	0

The new feed folder does not contain any items; it will do so when the time comes for Internet Explorer to refresh the feed's data. This is a setting for the browser and can be changed. We can force an update of the feed by requesting the feed URL in the browser:

```
PS> & 'C:\Program Files\Internet Explorer\iexplore.exe' $url
```

```
PS> dir 'PowerShell Blog'
```

Name	Title	Author	PubDate
0	Howto: Invoking ...	PowerShellTeam	9/26/2007 8:16:00 PM
1	"Notepad for Pow...	PowerShellTeam	10/9/2007 1:45:00 AM
2	IT administrator...	PowerShellTeam	10/17/2007 1:41:00 AM

```

3 Cool Stuff Coming! PowerShellTeam 10/27/2007 6:03:30 AM
4 Dynamic Casting PowerShellTeam 10/29/2007 6:14:22 PM
...

```

Once we have the items downloaded, we can read them like normal files. Remember that the name of the item is its numerical ID:

```

PS> cd ".\PowerShell Blog"
Microsoft Feeds\PowerShell Blog
PS> dir 14 | Format-List

```

```

LocalID : 14
Title   : Check it out: Out-vCard
Author  : PowerShellTeam
PubDate : 11/7/2007 11:51:51 PM

```

```

PS> (dir 14).Description
<p>Check out Dmitry Sotnikov's blog <a href="http://dmitrysotnikov.wo
rdpress.com/2007/11/07/out-vcard-exporting-outlook-address-book/">Out-
vCard: Exporting Outlook Address Book</a>.
</p><p>It lets you do things like:
...

```

When we are done, we can delete the subscription by deleting the feed folder we just created:

```

PS> del 'PowerShell Blog'

Confirm
The item at Feed:\Microsoft Feeds\PowerShell Blog has children and the
Recurse parameter was not specified. If you continue, all children
will be removed with the item. Are you sure you want to continue?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):Y

```

Manipulating Windows Active Directory

The DirectoryServices provider is a very valuable for system administration. PSCX will detect if you are running inside a Windows domain, and if so, it will register a new drive with the name of the domain. This is what that drive looks like for my test domain called TEST:

```

PS> get-psdrive | where { $_.Provider.Name -eq "DirectoryServices" }

Name      Provider      Root
-----
TEST      DirectoryS... TEST.local\

```

We can navigate the Active Directory objects using the file system abstraction:

```
PS> cd Test:\
TEST.local\
PS> dir
```

LastWriteTime	Type	Name
11/6/2007 10:06 AM	builtinDomain	Builtin
11/6/2007 10:06 AM	container	Computers
11/6/2007 10:06 AM	organizationalUnit	Domain Controllers
11/6/2007 10:06 AM	container	ForeignSecurityPrincipals
11/6/2007 10:06 AM	infrastructureUpdate	Infrastructure
11/6/2007 10:06 AM	lostAndFound	LostAndFound
11/6/2007 10:06 AM	msDS-QuotaContainer	NTDS Quotas
11/6/2007 10:06 AM	container	Program Data
11/6/2007 10:06 AM	container	System
11/6/2007 10:06 AM	container	Users

```
PS> cd Users
TEST.local\Users
```

```
PS> dir ADSITEST2
```

LastWriteTime	Type	Name
11/8/2007 5:21 AM	user	ADSITEST2

We can do recursive dir operations combined with a pipe through the where cmdlet to search for various objects. Here is how to get all groups in the directory:

```
PS> dir -R | where { $_.Type.Name -eq 'group' }
```

LastWriteTime	Type	Name
11/6/2007 10:38 AM	group	Account Operators
11/6/2007 11:07 AM	group	Administrators
11/6/2007 10:38 AM	group	Backup Operators
11/6/2007 10:06 AM	group	Distributed COM Users

...

We can also get various items' properties. For example, we can use the standard Get-ItemProperty cmdlet to display properties for our ADSITEST2 test user:

```
PS> cd \users
TEST.local\users
PS> Get-ItemProperty adsitest2
```



```

PSPath                : Pscx\DirectoryServices::TEST.local\users\adsitest2
PSParentPath           : Pscx\DirectoryServices::TEST.local\users
PSChildName            : adsitest2
PSDrive                : TEST
PSProvider              : Pscx\DirectoryServices
BadPasswordTime         :
BadPasswordCount        : 0
Department             :
DisplayName             : ADSITEST2
FirstName               : ADSITEST2
LastLogon               :
LastLogoff              :
LogonCount              : 0
Mail                   :
Manager                 :
OfficeName              :
PasswordLastSet         : 1/1/1601 2:03:37 AM
SamAccountName          : ADSITEST2
Surname                 :
Telephone               :
Title                   :
UserPrincipalName       : ADSITEST2@TEST.local
...

```

We can not only get information but modify our object, if we are running with domain administrator credentials. This is how we can set the Title and Mail attributes:

```

PS> Set-ItemProperty -path ADSITEST2 -Name Title -Value "Mr"
PS> Set-ItemProperty -path ADSITEST2 -Name Mail -Value "adsi@test.com"
PS> Get-ItemProperty -path ADSITEST2 -Name Title

```

```

PSPath                : Pscx\DirectoryServices::TEST.local\users\ADSITEST2
PSParentPath           : Pscx\DirectoryServices::TEST.local\users
PSChildName            : ADSITEST2
PSDrive                : TEST
PSProvider              : Pscx\DirectoryServices
Title                  : Mr

```

```

PS> Get-ItemProperty -path ADSITEST2 -Name Mail

```

```
PSPath      : Pscx\DirectoryServices::TEST.local\users\ADSITEST2
PSParentPath : Pscx\DirectoryServices::TEST.local\users
PSChildName  : ADSITEST2
PSDrive      : TEST
PSProvider   : Pscx\DirectoryServices
Mail         : adsi@test.com
```

Being a domain administrator allows us some power that we had better be careful with. This is how to delete a user from the Active Directory:

```
PS> del ADSITEST2
PS>
```

No, you do not even get a confirmation prompt! Be extra careful with that!

Note The DirectoryServices provider will work only if your machine is a member of a Windows domain. The drive that you can use to access the Active Directory objects will simply not appear otherwise.

Utility Applications

PSCX ships with two external applications that can make our lives easier when working with the shell. One of them is an advanced pager application that helps you read longer text, such as online documentation in a way that is superior to what the default `more` pager can offer. The other is a test utility that can help in debugging a call to an external application.

The first thing you notice after you install PSCX is that reading the help is a different experience. The default help function is changed to one that pipes the text through the `Less-394` program. It is a Windows port of the popular UNIX `less` pager. The first important feature that you will spot is the ability to go back to previously scrolled text; you can do that with the up arrow and Page Up keys. It looks like a little thing, but once you are used to it, you will never go back to the previous forward-only reading experience. The other nice feature is the ability to search for text. You can press the slash (/) key, type the search phrase, and press Enter. You will be transferred to the first match, and all matches will be highlighted. Figure 21-3 shows a search in progress; instances of the word “ComObject” are highlighted, because that is what was searched for.

You can press the N key to jump to the next search match. Press Shift+N to go to the previous match.

As much as I like the `less` pager, it has its shortcomings. First, it is not very intuitive to a Windows user. It has rather cryptic shortcut commands that are closer to the `vi` text editor shortcuts than to anything popular in the Windows world. You will have to do your portion of reading through the help file (press H to bring it up at any time) until you get the hang of the shortcuts. In addition, `less` suffers from a limitation that PowerShell has when dealing with input piped to legacy console applications. The text that is piped through `less` has to be collected in its entirety before being piped to the pager for display. This incurs a larger memory usage overhead besides the longer initial wait. Use `less` with extra caution when working with large portions of text.

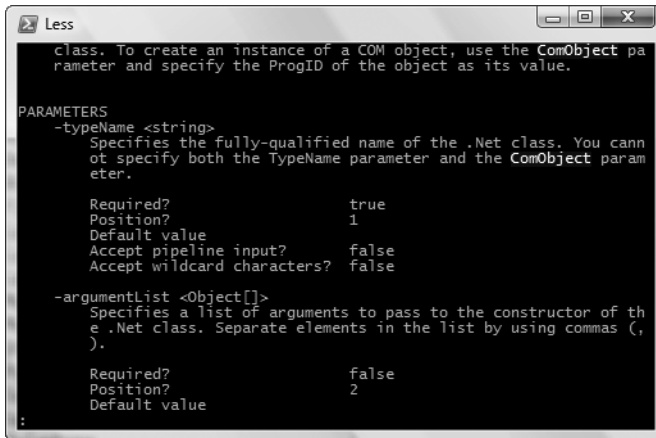


Figure 21-3. *The less pager performing an interactive search*

Another useful utility that ships with PSCX is the EchoArgs program. It is a small legacy console application that will just echo back to you the arguments it was given. It is useful when debugging legacy program invocations. Quite often, we think parameters are being passed to the program in the same way as if they were called from within `cmd.exe`, but that may not always be the case. Check out this program invocation:

```
PS> echoargs $a=b
Arg 0 is <=b>
```

You might be thinking that the `$` will get sent to the console program—not so. `$a` is treated as a variable reference, and that defaults to `$null`, which, in turn, is transformed into an empty string. The correct invocation has to escape the `$` symbol:

```
PS> echoargs `$a=b
Arg 0 is <$a=b>
```

Another common source of error is inadvertent terminator usage, usually the semicolon (`;`) and hash (`#`) symbols. The semicolon is treated as the statement end, and the hash turns the rest of the line into a comment. Here is a broken semicolon usage:

```
PS> echoargs do this; that; and that
Arg 0 is <do>
Arg 1 is <this>
```

The term 'that' is not recognized as a cmdlet, function, operable program, or script file. Verify the term and try again.

```
At line:1 char:23
```

```
+ echoargs do this; that; <<<< and that
```

The term 'and' is not recognized as a cmdlet, function, operable program, or script file. Verify the term and try again.

```
At line:1 char:28
```

```
+ echoargs do this; that; and <<<< that
```

Fixing this is simple—escape the semicolon:

```
PS> echoargs do this`; that`; and that
Arg 0 is <do>
Arg 1 is <this;>
Arg 2 is <that;>
Arg 3 is <and>
Arg 4 is <that>
```

The same holds true for the comment-start character. Here is a broken expression:

```
PS> echoargs look for item #3
Arg 0 is <look>
Arg 1 is <for>
Arg 2 is <item>
```

Again, the fix involves using the backtick escape character:

```
PS> echoargs look for item `#3
Arg 0 is <look>
Arg 1 is <for>
Arg 2 is <item>
Arg 3 is <#3>
```

Summary

PowerShell Community Extensions is a project that has accumulated a lot of useful functionality. What makes it really powerful is that it aims to collect pieces of code from the community—code that has been used by people to solve real problems. Chances are that that code will be useful to you should you have a similar problem.

Probably the most valuable thing about PSCX is that it is very comprehensive. With just a single installation, we get a very large toolset that removes many dependencies on other external tools. This makes our scripts easier to deploy across machines, as they would likely depend on only one product instead of many. Using a single large toolset is also easier to gain approval for in a corporate environment with strict policies on what software gets installed on workstations and servers. I highly recommend this tool—I routinely install it on a new machine right after PowerShell installation.



PSEventing: .NET Events in PowerShell

PowerShell gives you full control over all .NET objects—well, nearly full control. One often-missed feature is full support for consuming events. It is possible to subscribe to an event handler using PowerShell constructs; script blocks can be passed to .NET objects as delegates. The problem is that there are a lot of complexities involved in this process like thread synchronization and garbage collection—someone needs to manage object life spans so that events going across scripting boundaries do not find that their handler has been garbage collected. Additionally, we need some orchestration when it comes to raising events from multiple threads. PowerShell does not offer any intrinsic support for those scenarios, and you can easily crash the entire shell if you are not careful in how you implement them.

Lacking full support for events may look like a big omission, but the truth is that, most of the time, we do not need to deal with events at all. Most scripts are short-lived, short-running, or both. Nobody builds event-driven solutions or implements complex application frameworks in shell scripts, right? There are better tools for full-blown application development—please use C# or VB.NET if you have to deal with those types of issues! Shell scripting is really about gluing components and working at a higher level of abstraction.

Rant aside, sometimes, we can benefit from being able to consume a .NET class's events from script. The most obvious use for that is a script that sits and waits for a `FileSystemWatcher` object to notify it when a user deletes a file from a folder. This is where the PSEventing tool comes to the rescue.

Introducing PSEventing

PSEventing is a free, open-source project run by Oisín Grehan and hosted on CodePlex at <http://www.codeplex.com/PSEventing>. It is implemented as a shell snap-in that provides several cmdlets for working with events.

Although the current release, version 1.0, does not offer an automated installer, installation is relatively simple. Here are the steps you need to take:

1. Unzip the distribution in the folder of your choice and navigate to it. A good location might be `C:\Program Files\Windows PowerShell Eventing`.
2. Run the `install.ps1` script contained in the distribution. The script will invoke `InstallUtil.exe` and install the snap-in.

Note Each PowerShell snap-in inherits from the `PSSnapIn` class, which, in turn, inherits from `PSSnapInInstaller`. `PSSnapInInstaller` inherits from the .NET Framework `Installer` class and plugs into the standard .NET application installation infrastructure. Being an installer, a snap-in, among other things, reads and writes data to the system registry. That is why you have to run the `install.ps1` script under an administrator user account.

An important thing to keep in mind is that “installing” a snap-in only registers it with PowerShell. It does not automatically run it whenever the shell starts. We have to add the snap-in to our shell session before using any of the new cmdlets. To do that, we use the `Add-PSSnapin`. The command we need for registering `PSEventing` is simple:

```
Add-PSSnapin PSEventing
```

Note The `Add-PSSnapin` cmdlet invocation is not needed if you are still running the same shell session that you used to install `PSEventing`. That shell is run with elevated privileges, and it is best, for security reasons, if you exit it and start a shell with normal privileges.

Writing scripts that use `PSEventing` presents us with an interesting challenge: how do we know if the snap-in is loaded already, so that we do not load it twice? We will get an error if we try to load the snap-in for the second time. A possible solution is to use `Get-PSSnapin` to check that:

```
if ((Get-PSSnapin PSEventing -ErrorAction SilentlyContinue) -eq $null)
{
    Add-PSSnapin PSEventing
}
```

That feels like the cleanest solution, but I usually prefer the lazy person’s alternative:

```
Add-PSSnapin PSEventing -ErrorAction SilentlyContinue
```

The preceding line just ignores any error that may be raised while adding the snap-in. Of course, a load error may have many causes, and ignoring it may not be very smart. Most of the time, `Add-PSSnapin` will complain about loading something twice, so I feel pretty safe.

Event-Handling Mechanics

An event in .NET is usually given a delegate, and the sender invokes that delegate when it raises the event. Many people are used to that paradigm and would probably expect that that would be the case with `PSEventing`. Maybe we can just pass a script block as an event handler—not so fast! PowerShell lacks good threading support for scripts, and firing an event from a foreign piece of code may be dangerous from a concurrency point of view.

That is why `PSEventing` takes a slightly different approach: it creates the abstraction of a global event queue. All access to the event queue will be synchronized to protect us from

concurrency-related problems. The rules according to which the queue is filled are stored in the event binding table. You call `Connect-EventListener` to connect to an event and that creates an entry in the binding table. You can use the `Get-EventBinding` cmdlet to inspect that table. A sample table that shows two entries follows:

```
PS C:\> Get-EventBinding
```

VariableName	EventName	TypeName	Listening
-----	-----	-----	-----
fsw	Created	FileSystemWatcher	True
fsw	Deleted	FileSystemWatcher	True

Here, we have two bindings that will handle the `Created` and `Deleted` events of the `fsw` variable, which is of type `FileSystemWatcher`. Unbinding an event is done via the `Disconnect-EventListener` cmdlet.

Binding to an event does not involve any way of specifying a callback that will be executed when the event is raised. When an event is raised, it is added to the queue, and the rest is left up to the client code. You are free to poll the queue as often as you wish and pull the events out of it. The cmdlet that queries the queue and consumes all events is `Get-Event`. It returns `PSEvent` objects, from which you can get the event arguments, passed by the event sender, by querying the `Args` property. A special feature of `Get-Event` is that it can block execution and wait for an event to be added to the queue. This spares us the trouble of writing polling loops!

FileSystemWatcher: A Real-World Example

Let's go through a complete example and put everything together. Assume we are presented with the task of monitoring a folder and notifying the user when somebody deletes a file. To do that, we use a `System.IO.FileSystemWatcher` object and subscribe to its `Deleted` event. We will do that from a script called `WatchDeletedFiles.ps1`:

```
#monitor deleted files
```

```
Add-PSSnapin PSEventing -ErrorAction SilentlyContinue
```

```
$fsw = New-Object System.IO.FileSystemWatcher
$fsw.Path = "c:\powershell"
$fsw.EnableRaisingEvents = $true
```

```
Connect-EventListener fsw deleted
```

```
Get-Event -wait | `
    foreach { `
        Write-Host -foreground Yellow `
            "Warning!!! Somebody just deleted $($_.Args.FullPath)"
    }
```

```
Disconnect-EventListener fsw deleted
$fsw.EnableRaisingEvents = $false
```

We are monitoring the `C:\powershell` folder, and the only `FileSystemWatcher` event of interest is the `Deleted` one. Keep in mind that there can be more than one event in the queue, for example, when a user deletes multiple files in one operation. That is why we use `For-EachObject` on the `Get-Event` output. Deleting files is some serious business, so we emit shiny yellow text to get the user's attention.

We pay extra attention to cleaning up before the script exits. We do not need to listen to the event anymore, and we do not want to leave garbage entries in the global event binding table, so we just go ahead and disconnect from the event. In addition, we tell the `FileSystemWatcher` object, which the `$fsw` variable references, to stop raising events. Running the script will block the console session until a file is deleted. When that happens, the full file path will be logged to the console, and the script will terminate:

```
PS C:\PowerShell> .\WatchDeletedFiles.ps1
Warning!!! Somebody just deleted c:\powershell\VeryImportant.dll
```

Monitoring Entries Written to the System Event Logs

.NET allows us to access the system event logs, provided we have been granted sufficient privileges. We can do that through the `System.Diagnostics.EventLog` class that is a part of the framework. That class is a very powerful tool and can be used to write entries to the event log, and I covered using it in Chapter 16. Most documentation resources about that class focus on its ability to write entries to an event log. We are interested in another feature: the ability to receive notifications about newly written entries. The class exposes an `EntryWritten` event that will be fired not only when our application logs an event but whenever a different application, running on the same computer, does so. The event is limited to working on only the local machine and will not fire if you are working with a remote event log, but operating on the local machine is enough in many cases.

As an example, we can create a script that waits on the `EntryWritten` event and notifies us when a user logon is attempted. We will be receiving instances of the `System.Diagnostics.EntryWrittenEventArgs` object, and we will access the new `EventLogEntry` object via the `Entry` property: `$_Args.Entry`. Additionally, we need to filter events, since we are interested in only the logon attempt events. We filter using the `InstanceId` property: we are looking for entries with an instance ID of 4648. How do we get the correct number for our event? Simple—we just look it up in the system event viewer. Most likely, we have a previously logged event in there. The event log entry will contain the instance ID, but it will call it the `Event ID` property. This discrepancy comes from the fact that “Event ID” is the obsolete name of the instance ID property. Figure 22-1 shows a sample view of a logon event.

Note that the 4648 event that we are using here is a security audit event; its source is `Security-Auditing`. Windows Vista has its user logon auditing turned on by default, but you may need to enable it on your system.



Figure 22-1. An event log entry showing a Logon audit event (Event ID = 4648)

Finally, here is the script, WatchEventLog.ps1, that listens for the event:

```
Add-PSSnapin PSEventing -ErrorAction SilentlyContinue

$securityLog = New-Object System.Diagnostics.EventLog "Security"
$securityLog.EnableRaisingEvents = $true

Connect-EventListener securityLog EntryWritten
$logonAttemptEventInstanceID = 4648

Get-Event -wait | `
    where { `
        $_.Args.Entry.InstanceID -eq $logonAttemptEventInstanceID
    } | `
    foreach { `
        Write-Host -foreground Red `
            "Logon attempt at: $($_.Args.Entry.TimeGenerated)"
    }

Disconnect-EventListener securityLog EntryWritten
$securityLog.EnableRaisingEvents = $false
```

The script filters the events according to the InstanceID property and then notifies the user by writing out in red the time the event was generated. Note that, just as with the FileSystemWatcher example, we need to disconnect our event listener and tell the EventLog object to stop raising events as a part of our clean up logic.

A word of caution about accessing the Security event log—it will require administrator privileges on any system. Running the script on Windows Vista may result in the following error:

```
C:\PowerShell> .\WatchEventLog.ps1
Exception setting "EnableRaisingEvents": "Requested registry access is
not allowed."
At Z:\22 - PSEventing\WatchEventLog.ps1:4 char:14
+ $securityLog.E <<<< nableRaisingEvents = $true
```

The error is quite descriptive; the script lacks the required privileges. To make it work, ensure that your PowerShell session is running with elevated privileges. To do so, right-click the PowerShell shortcut, and select the “Run as an administrator” item from the context menu.

Here is what happens after running the script and logging on with a different user account:

```
PS C:\PowerShell> .\WatchEventLog.ps1
Logon attempt at: 07/15/2007 09:01:59
```

Handling WMI Events

WMI is a complex and powerful tool for managing a Windows network. It is the infrastructure that provides common interfaces to managing operating system devices, services, and applications. PowerShell already has excellent support for querying WMI objects and extracting information about various system components. For details on how to do that, please refer to Chapter 20. With PSEventing, we can go further and handle events that WMI triggers when it detects a change in the operating system environment. The .NET classes from the System.Management namespace, most importantly the ManagementEventWatcher one, are the means of receiving notifications about those changes. The ManagementEventWatcher class is a generic listener; it needs a separate query object that specifies the type of event we are interested in. Additionally, the query can specify filter conditions based on the event source’s properties and a poll interval. Queries are written in the WQL language that is very similar to SQL. Querying the WMI system looks like working with a database table. To create something useful, we can consume WMI events that tell us when the World Wide Web Publishing Service (W3SVC) service stops. Here is our query that will look for the instance modification event about stopping the W3SVC service:

```
SELECT *
FROM __InstanceModificationEvent
WITHIN 10
WHERE
    TargetInstance ISA 'Win32_Service'
    AND TargetInstance.Name = 'w3svc'
    AND TargetInstance.State = 'Stopped'
```

We are getting objects from the special `__InstanceModificationEvent` table, and we are interested in Windows services that have a name of “w3svc” and a status of “Stopped.” The `WITHIN` operator specifies the interval in seconds in which the system will poll for updates. A WMI query is a resource-intensive operation, and we have chosen to poll every 10 seconds, so that we do not overload the system. Here is the script, `Watch_w3svc.ps1`, that listens to the event watcher’s generic `EventArrived` event to do the job:

```
Add-PSSnapin PSEventing -ErrorAction SilentlyContinue

$queryString = @"
    SELECT *
    FROM __InstanceModificationEvent
    WITHIN 10
    WHERE
        TargetInstance ISA 'Win32_Service'
        AND TargetInstance.Name = 'w3svc'
        AND TargetInstance.State = 'Stopped'
"@

$query = New-Object System.Management.WQLEventQuery `
    -argumentList $queryString
$watcher = New-Object System.Management.ManagementEventWatcher($query)

Connect-EventListener watcher EventArrived
$watcher.Start()

echo "Waiting for the W3CSVC service to stop..."
Get-Event -wait | `
    foreach { `
        Write-Host -foreground Red "The W3SVC service has stopped!"
    }

#cleanup
$watcher.Stop()
Disconnect-EventListener watcher EventArrived

echo "done"
```

We create a `WQLEventQuery` object using our WQL query string. We then pass the query object to the `ManagementEventListener` object’s constructor. An important thing to keep in mind is that we will never get an event until we call the event watcher’s `Start` method. When we are done, we call the `Stop` method in order to let the event watcher clean up properly. We then disconnect from the `EventArrived` event. Here is what our script’s output looks like after running it and stopping the W3SVC service:

```
PS C:\PowerShell> .\Watch_w3svc.ps1
Waiting for the W3CSVC service to stop...
The W3SVC service has stopped!
```

The preceding WMI events script is very useful, but it can be very dangerous too! Failing to call the event watcher `Stop` method will leave the system executing the query over and over again. Running the script several times can cause event watchers to add up, and our system can slow down significantly because most of the CPU time will be allocated to executing leaked queries. Remember to stop your event watchers and to disconnect from their events when you are done with them!

Detecting If Our Script Has Been Terminated by the User

Let's get back to our first example, which watches a folder for file delete operations, and see what happens if the user presses `Ctrl+C` while the script is listening for events. We will add a `Write-Host` call after the script that will get called after the script ends. Here is what we get:

```
PS C:\PowerShell> .\WatchDeletedFiles.ps1 ; Write-Host "Done"
PS C:\PowerShell>
```

Yes, we get nothing—not even the “Done” message. PowerShell terminates not only the script but the entire command that has been typed. Handling `Ctrl+C` is a weak spot in PowerShell 1.0, and that is why the `PSEventing` snap-in provides two cmdlets that can help us: `Start-KeyHandler` and `Stop-KeyHandler`. They can trap various key presses, but we are interested in handling `Ctrl+C` only. I know, learning a new programming environment requires writing your own Tetris clone, and you certainly can do that with those cmdlets, but I will not be covering it in this book.

Those cmdlets allow us to write scripts that will not be terminated when the user presses `Ctrl+C`. Now, the scripts can react accordingly and most likely release resources properly before actually terminating. The key (pun intended) to supporting `Ctrl+C` is to

1. Register a handler using `Start-KeyHandler -CaptureCtrlC`.
2. Distinguish `Ctrl+C` events from the rest.
3. Unregister the handler when you are done by calling `Stop-KeyHandler`.

Of the three, distinguishing the events is the only one that is not immediately obvious. We will use the fact that `Ctrl+C` events will have a `Name` of “`CtrlC`”, as they are returned by `Get-Event`. Here is a modified version of our folder watcher script, called `WatchDeletedFilesCtrlC.ps1`, which handles `Ctrl+C` correctly:

```
Add-PSSnapin PSEventing -ErrorAction SilentlyContinue
```

```
$fsw = New-Object System.IO.FileSystemWatcher
$fsw.Path = "c:\powershell"
$fsw.EnableRaisingEvents = $true
```

```

Start-KeyHandler -CaptureCtrlC
Connect-EventListener fsw deleted

do
{
    $events = Get-Event -wait
}
while(!$events)

foreach ($event in $events)
{
    if ($event.Name -eq "CtrlC")
    {
        Write-Host "Ctrl+C detected"
        break;
    }
    else
    {
        Write-Host -foreground Yellow `
            "Warning!!! Somebody just deleted $($event.Args.FullPath)"
    }
}

Stop-KeyHandler
Disconnect-EventListener fsw deleted
$fsw.EnableRaisingEvents = $false

```

This time, we are looping over the events using a `foreach` loop and using `if` statements inside, which is semantically equivalent to the `where` cmdlet we used before. The `where`-based solution was easier to use when we needed to filter events according to one condition. Here, we have two types of events we need to handle. Note the `do-while` loop that we use around our `Get-Event` call. This is needed because sometimes `Get-Event` will return a `$null` value after a Control-C event. The loop just ignores that value and starts another event-listen operation. Our new script will only warn the user if the file has really been deleted.

Here is what happens when we run our script—it will not be terminated, and the shell will continue executing the script and the command on the same line:

```

PS C:\PowerShell> .\WatchDeletedFilesCtrlC.ps1 ; write-host "Done"
Ctrl+C detected
Done
PS C:\PowerShell>

```

Using Script Blocks As Event Handlers

The previous example showed something nice, but it also hinted about something that is not so pleasant: it is very difficult to deal with different events and distinguish among them. The

best way to handle this problem is to use the helper `eventhandler.ps1` script that ships with PSEventing and its utility functions:

- `Add-EventHandler($variable, $eventName, $script)` will attach a script block as a handler for the given event.
- `Remove-EventHandler($variable, $eventName)` lets us detach a previously attached event handler.
- `Do-Events($onlyOnce)` will wait for events and process their event handlers one or more times.

The `eventhandler.ps1` file is a part of the official PSEventing release files, but it is not included in the ZIP archive that contains the snap-in assembly and installer script. Make sure you download it and place it in the PSEventing installation folder. The following example uses the standard location of `C:\Program Files\Windows PowerShell Eventing`.

Let's expand our file system watcher example and make it handle both the Deleted and Created events. This is best done with two calls to `Add-EventHandler`—one for each event type—followed by a `Do-Events` call. We will call that script `WatchDeletedCreatedFiles.ps1`. Here is the code:

```
Add-PSSnapin PSEventing -ErrorAction SilentlyContinue
#include the PSEventing utility function definitions
. "C:\Program Files\Windows PowerShell Eventing\eventhandler.ps1"
```

```
$fsw = New-Object System.IO.FileSystemWatcher
$fsw.Path = "c:\powershell"
$fsw.EnableRaisingEvents = $true

$fswVariable = (Get-Variable fsw)
$deletedBlock = {
    param ($sender, $args)
    out-host -input "file deleted"
    format-list -property ChangeType,FullPath -input $args
}
$createdBlock = {
    param ($sender, $args)
    out-host -input "file created"
    format-list -property ChangeType,FullPath -input $args
}
```

```
Add-EventHandler $fswVariable Deleted $deletedBlock
Add-EventHandler $fswVariable Created $createdBlock
```

```
Do-Events $true
```

```
Remove-EventHandler $fswVariable Deleted
Remove-EventHandler $fswVariable Created
$fsw.EnableRaisingEvents = $false
```

The script uses the `Format-List` cmdlet to display details about the event it got. Here is what the output looks like when we run the script, create a copy of a file in our target folder, and delete it afterward:

```
PS C:\PowerShell> .\WatchDeletedCreatedFiles.ps1
file created
```

```
ChangeType : Created
FullPath   : c:\powershell\test - Copy.txt
PS C:\PowerShell>
PS C:\PowerShell> .\WatchDeletedCreatedFiles.ps1
file deleted
```

```
ChangeType : Deleted
FullPath   : c:\powershell\test - Copy.txt
```

Our script calls `Do-Events` without parameters. This triggers the default behavior: the function will wait forever or until it receives a Ctrl+C key press. If we don't want that, we need to pass a `$true` parameter like this:

```
Do-Events $true
```

The nicest thing about `Do-Events` is that it allows us to quickly create utilities that sit in the background and notify us about events that occur on our system. It spares us the trouble of setting up a while loop that calls `Get-Event` repeatedly.

Summary

Handling events from .NET objects is not the primary usage scenario for PowerShell, which could explain why the samples we saw in this chapter look complex and feel like ugly hacks. Being able to use and create similar solutions requires knowledge about the .NET Framework and WMI, and a programmer or a system administrator will either have that knowledge or will greatly benefit from acquiring it. There is great value in being able to work reliably with events from within scripts, and having a tool like `PSEventing` under your belt can really help you solve an otherwise unsolvable scripting problem.



Enhancing Tab Completion with PowerTab

I remember seeing tab completion at work for the first time in a bash shell session running under Linux several years ago. A friend was typing a really long file name in a bash shell session. I had already started complaining that he should have copied and pasted the file name when he hit the Tab key—and the shell automatically completed the entire name for him. My jaw hit the floor. I had been using DOS and the Windows command prompt for years before that, and I never imagined the productivity gains I was missing.

Later versions of Windows, starting with Windows 2000, have added rudimentary support for tab completions to the `cmd.exe` command shell, and things have gotten a bit better. PowerShell is the first product that offers good completion support that works for files, functions, variables, and commands and can be customized if needed. I already discussed how tab completion works in Chapter 11, and you even learned how to define your own expansion functions to provide support for completing custom executable names. PowerTab takes that route and adds so many advanced features that even UNIX shell users will want to switch to PowerShell.

PowerTab is a free tool that has been built by Marc van Orsouw, also known as `/\o\` or The PowerShell Guy. The tool consists of a set of scripts that plug into PowerShell's mechanism for extending the tab completions offered to the user. It tries to provide better completion suggestions by searching objects that the default shell behavior does not even touch: .NET types, WMI class names, and many more. The tool also improves the user experience by providing a drop-down box that contains the suggestions. The default tab expansion allows us to cycle through the suggestions by pressing the Tab key while PowerTab lists all suggestions and allows us to select the desired one using the arrow keys or the Tab and Shift+Tab keys.

I have always believed that starting to use an extension tool while learning a new product is not a good idea. For that reason, I was reluctant to install add-ons and custom scripts while I was learning PowerShell, and I viewed PowerShell's default tab completion support as sufficient for my needs. One afternoon, I installed PowerTab, just to try it out, and I have never looked back. The tool helps me so much when working with .NET and WMI objects that I cannot imagine using PowerShell without it anymore. In this chapter, we will go through PowerTab's major features, which can make your life at the command prompt much easier.

Installation

PowerTab does not come with an automated installer. Since getting it installed and running doesn't require much effort, it is available in a simple ZIP archive from the author's site at <http://thepowershellguy.com/blogs/posh/pages/powertab-beta.aspx>. As you can see from the URL, at the time of this writing, PowerTab is still in beta version. The latest version of the product is 0.99 beta 2. The tool is remarkably stable, though, and I have not seen it crash or misbehave in any way. So, without waiting any longer, here is how to install it:

1. Download the ZIP distribution from the beta page. The current file name is `PowerTab099b2.zip`.
2. Extract the archive contents to a folder of your preference. Using a standard location like `C:\Program Files\PowerTab` is best.
3. Launch PowerShell, and navigate to the folder you installed PowerTab in.
4. Execute the `PowerTabSetup.ps1` script by dot-sourcing it; the command should be

```
PS> . .\PowerTabSetup.ps1
```

```
PowerTab 0.98 PowerShell TabExpansion Setup
```

```
...
```

5. Answer the questions the install script presents, and then wait for it to do its job.

Note Dot-sourcing the script in step 4 is not strictly necessary, but it is required to get PowerTab running for the current shell session. If you do not dot-source the script, you will need to restart your shell instance.

The setup script will create a default configuration file. The file is in a readable XML format and is called `PowerTabConfig.xml`. By default, it is stored next to your profile script.

Next, the script will update your profile script, and it will append some code that activates PowerTab whenever you start your shell. The code will look like this:

```
##### Start of PowerTab Initialisatie Code #####
#
# added to Profile by PowerTab Setup For Loading of
# Custom TabExpansion,
#
# /\o\ 2007
#
# http://ThePowerShellGuy.com
#

# Initialize PowerTab

& 'C:\Program Files\PowerTab\Init-TabExpansion.ps1' `
-ConfigurationLocation 'C:\Users\Hristo\Documents\WindowsPowerShell'
```

Finally, the setup script will build the tab completion database and save it to the `TabExpansion.xml` file. Building the database is a lengthy process, as the script walks over all .NET types and WMI classes. The end result is a local cache with class and command names that are used for providing tab completions.

How PowerTab Works

PowerTab completely replaces the shell's tab completion mechanism. It supplies all the standard completions that you are used to: commands, variables, functions, and so on, and adds a lot more. For example, I find PowerTab very useful when I am trying to script a .NET type. Remember the code that used the `System.Net.WebClient` class from Chapter 17? PowerTab can help you by completing the type name in the `New-Object` cmdlet invocation line if you press `Tab` while typing the full class name. Finish typing the namespace part, at least, so that you do not get a huge list of suggestions. Figure 23-1 shows how the suggestions list looks.

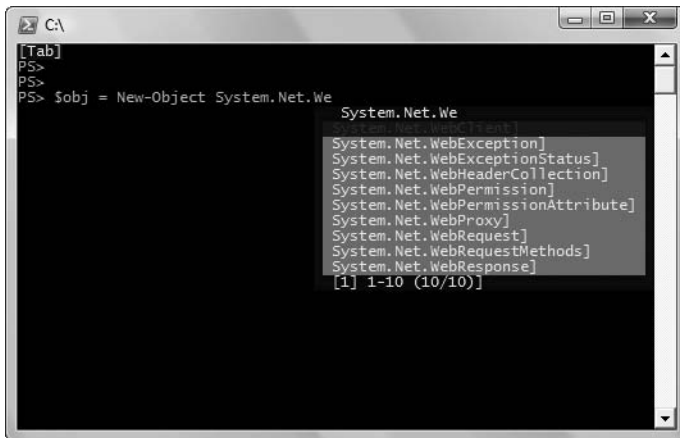


Figure 23-1. *PowerTab completing .NET class names*

See the `[Tab]` text shown at the upper-left corner of the shell window? It indicates that PowerTab is in the middle of a tab completion operation. It will go away once you select a completion from the list.

When it comes to working with various objects, no matter if they are coming from the .NET, COM, or WMI realm, PowerTab will enumerate their properties and methods and will offer completions. Continuing with our `System.Net.WebClient` example, PowerTab can help us download data from the Internet by completing the right `Download*` method for us if we start typing the method name and press `Tab`. The result is shown in Figure 23-2.

Method and property completion works not only with instance methods but with static methods too. PowerTab detects the type literal and the `::` separator and offers completions from the type's static members.

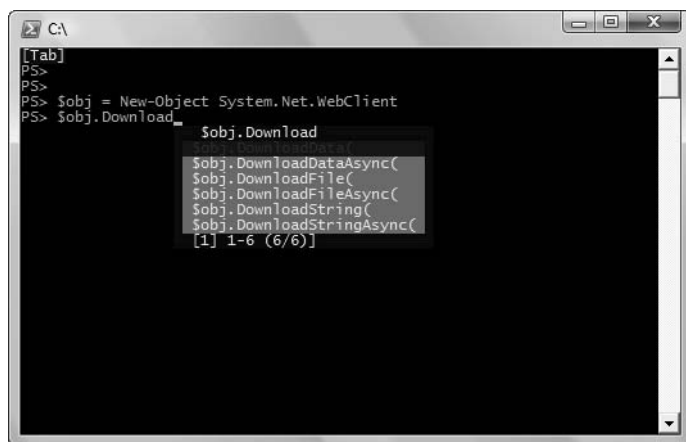


Figure 23-2. *PowerTab completing object methods*

I happen to write a lot of WMI-related code, and I believe that the PowerShell environment makes working with WMI as easy as it can get. PowerTab makes that even easier by providing completions for WMI class names. One of the biggest hurdles when working with WMI is remembering the right class that you have to use because of the huge number of classes. PowerTab finally solves that problem. Figure 23-3 shows how PowerTab can assist us with picking the right WMI class for working with processes.

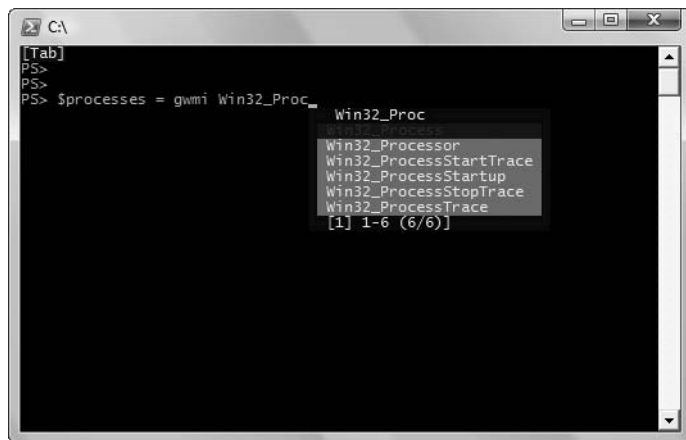


Figure 23-3. *PowerTab completing WMI class names*

PowerTab detects WMI class names from their prefix, and the completion kicks in if the current word starts with WMI_, CIM_ or MSFT.

I have always found the command history support in PowerShell a bit lacking. Sure, we can always use DOSKEY and press F7 anytime to get a nice visual list of all the commands, but that list is not searchable and gets hard to navigate when we have a big history. PowerTab solves that problem by introducing special syntax for searching the history buffer and providing completions. If we need to get all commands containing the “cd” string, we need to type **h_cd**

and press Tab. Figure 23-4 shows the result—all commands that have been used to navigate to a different folder.

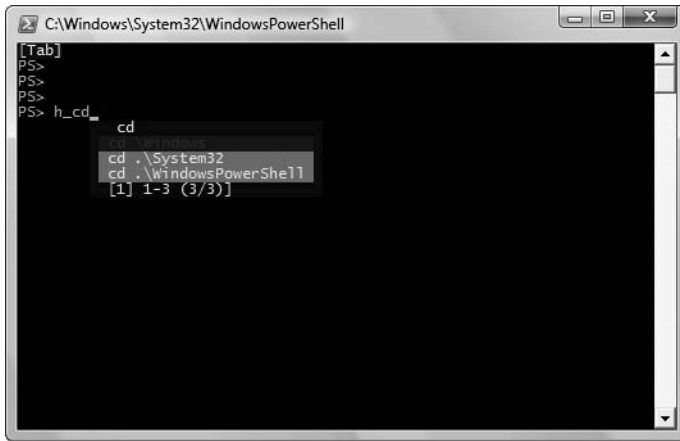


Figure 23-4. *PowerTab command history completion*

The `h_` prefix that you need to type may seem like it would be appended to the final command text. Do not worry about that happening. Unlike other completions that simply add to the current command, this one will also delete the `h_` part. Of course, typing only `h_` and pressing Tab will get you a list of all commands stored in the history buffer.

Command parameter names are also very hard to remember when working with PowerShell. For example, I keep trying to use a nonexistent `-Process` parameter when getting process instances with the `Get-Process` cmdlet, where I should be using the `-Name` one. I guess I am not the only person that has this problem, as PowerTab has a feature that solves it. Whenever you type a hyphen after a cmdlet name and press Tab, you will get a completion list with all its parameters. Figure 23-5 shows the completion list for the `Get-Process` cmdlet.

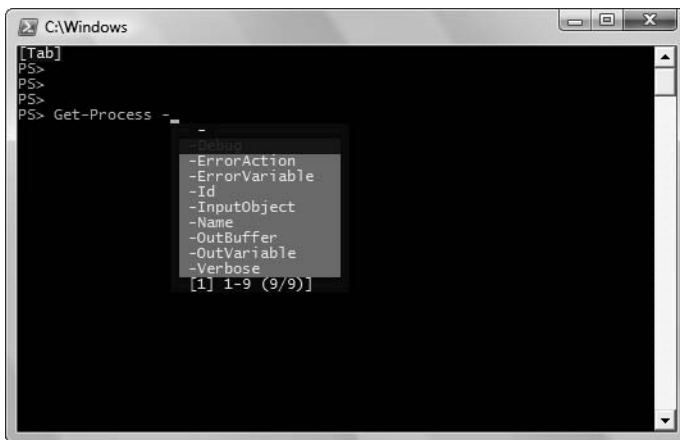


Figure 23-5. *PowerTab parameter name completion*

The best part about this feature is that it works for functions and external script files too.

Data Grid Pop-up Windows

Sometimes, cramming lots of information in that tiny box that appears inside the shell window gets hard. PowerTab tries to deal with that by opening up a new window. The window blocks the tab completion process and displays a data grid. The user can get detailed information about the objects in the grid and select one by double-clicking a row or selecting it and pressing Enter. The selected item gets returned as the valid completion.

Similar to the syntax for history completions, PowerTab offers several shortcuts that pop up a new window. Probably the most attractive is the one that returns a list of WMI classes. To get the window, we have to type `w_` optionally followed by a part of a WMI class name and press Tab. Figure 23-6 shows the window displaying information about the WMI classes considered for completion.

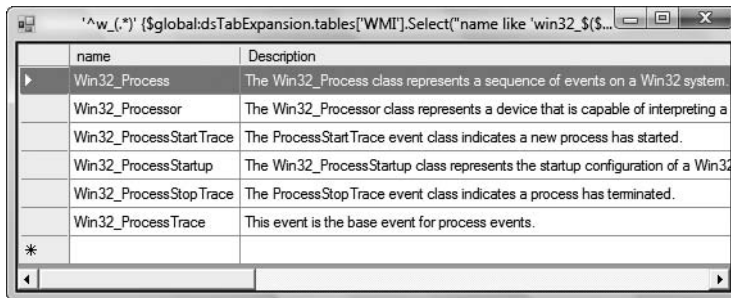


Figure 23-6. PowerTab pop-up grid window displaying WMI classes

I like this completion method better, especially when it comes to WMI classes, because we get a Description column with a short explanation of what a class does.

Note You have to skip the `Win32_` part of a WMI class name when using the `w_` shortcut to request completions with a pop-up window. To get classes that look like the `proc` string, possibly looking for `Win32_Process`, you have to type `w_proc[tab]`. Typing `w_win32_proc[tab]` will not work.

PowerTab supports other shortcuts that pop up a new window:

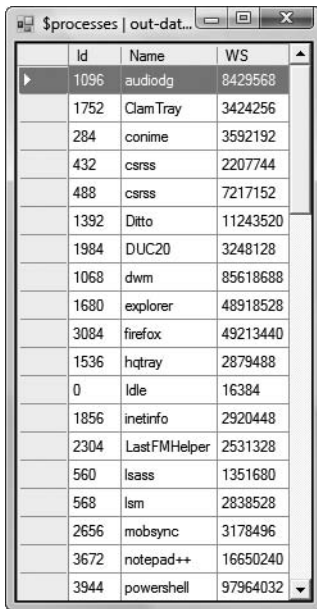
- `t_`: Shows a list of .NET types
- `f_`: Lists currently defined functions
- `d_`: Gets all files and directories in the current directory
- `g_`: Lists commands stored in the command history
- `c_`: Shows custom completions (I describe adding custom tab completions later in this chapter.)

A Useful Utility

PowerTab makes available the function that creates the pop-up window to us as users, so that we can display objects in a bit more user-friendly manner. The name of the function is `out-dataGridView`, and we can pipe an object collection to it. Here is how to use it to display the process ID, name, and working set property values for all processes:

```
PS> $processes = Get-Process | select ID,Name,WS
PS> $processes | out-dataGridView
```

Figure 23-7 shows the resulting window.



Id	Name	WS
1096	audiodg	8429568
1752	ClamTray	3424256
284	conime	3592192
432	csrss	2207744
488	csrss	7217152
1392	Dllho	11243520
1984	DUC20	3248128
1068	dwm	85618688
1680	explorer	48918528
3084	firefox	49213440
1536	hqtray	2879488
0	Idle	16384
1856	inetinfo	2920448
2304	LastFMHelper	2531328
560	lsass	1351680
568	lsm	2838528
2656	mobsync	3178496
3672	notepad++	16650240
3944	powershell	97964032

Figure 23-7. A grid with all processes on the system generated by `out-dataGridView`

Configuring PowerTab

The heart of the PowerTab configuration is the `PowerTabConfig.xml` file. It is stored next to your PowerShell profile script; under Windows Vista, that is most likely the `C:\Users\<UserName>\Documents\WindowsPowerShell` folder. The XML content is pretty readable, and it is very easy to find your way around the file and to edit it by hand. Settings are exposed in the form of properties, and their XML elements look like this:

```
<Config>
  <Category>Global</Category>
  <Name>TabActivityIndicator</Name>
  <Value>1</Value>
  <Type>bool</Type>
</Config>
```

As you can probably guess by looking the preceding snippet, you need to change the 1 to 0 if you want to keep PowerTab from showing the [Tab] text at the top-left corner of the shell window while displaying tab expansions. Remember to restart your shell session after you modify the configuration file.

PowerTab allows for modification of its configuration on the fly. It exposes a global `$PowerTabConfig` variable that is the entry point for all configuration settings. For example, finding out if the tab activity indicator is enabled is a matter of getting the `TabActivityIndicator` property:

```
PS> $PowerTabConfig.TabActivityIndicator
True
```

Disabling the indicator is a matter of setting the property to `$false`:

```
PS> $PowerTabConfig.TabActivityIndicator = $false
PS>
```

That is all we need. The activity indicator will not bother us anymore. The only problem with modifying the configuration on the fly is that it does not get persisted. The next shell session will have the original settings. To save the configuration, we need to call the `Export-tabExpansionConfig` function:

```
PS> Export-tabExpansionConfig
Configuration exported to C:\Users\Hristo\Documents\WindowsPowerShell\PowerTabConfig.xml
```

The IntelliSense Completion Handler

PowerTab has the concept of a tab completion handler. The default one, the one we have been using until now, is the `ConsoleList` handler. It is the function that knows how to draw the nice box that contains the completion list. PowerTab ships with a PowerShell snap-in, the `Lerch.PowerShell.dll` file, that provides another completion handler—the `intellisense` one. The handler is named that way, because it opens a small pop-up window that looks like the `IntelliSense` pop-ups that Visual Studio .NET and other development environments use to provide completions. The author of the snap-in is Aaron Lerch (<http://www.aaronlerch.com>).

To enable the `intellisense` handler, you first need to register the `LerchSnapIn` snap-in. To do that, start PowerShell as an administrator, visit the PowerTab installation folder, and invoke `installutil.exe` on the `Lerch.PowerShell.dll` file:

```
PS> cd "\Program Files\PowerTab"
C:\Program Files\PowerTab
PS> installutil .\Lerch.PowerShell.dll
Microsoft (R) .NET Framework Installation utility Version 2.0.50727.312
```

Copyright (c) Microsoft Corporation. All rights reserved.

...

The Commit phase completed successfully.

The transacted install has completed.

Make sure that the installation completed successfully and that you do not see any errors reported by the `installutil` command.

Note The `installutil.exe` program is part of the .NET Framework. If it is not already available in your system PATH, you can find the executable file inside the framework installation folder. The folder is typically located at `C:\Windows\Microsoft.NET\Framework\v2.0.50727\`.

If the installation went smoothly, you need to make sure that the snap-in is always loaded in your shell session. To do that, you have to add a call to `Add-PSSnapIn` to your profile script. Open your profile script, most likely the `C:\Users\<UserName>\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1` file, and add the following lines before the PowerTab initialization code:

```
#add the Lerch IntelliSense snap-in (needed by PowerTab)
Add-PSSnapin LerchSnapIn
```

The comment is not really necessary, but it will help you remember why the line is there when you open your profile script in six months.

Now, you are ready to modify the PowerTab configuration. You need to set the `DefaultHandler` property:

```
PS> $PowerTabConfig.DefaultHandler = 'intellisense'
```

That is all. Try pressing Tab for completions. You should get a small pop-up window that looks similar to the one shown in Figure 23-8.

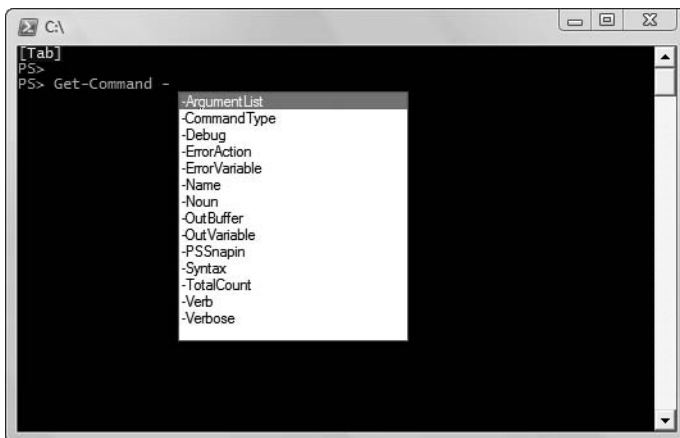


Figure 23-8. The *intellisense* completion handler showing *Get-Command*'s parameter list

Remember to call the `Export-tabExpansionConfig` function; otherwise, when you restart the shell, you will get the `ConsoleList` handler back.

Pressing Tab Twice

PowerTab can detect if you have quickly pressed the Tab key twice. If you do, you can bind a different completion handler that will get invoked by setting the `AlternateHandler` configuration property. This way, you can have the `ConsoleList` handler bound to the Tab key and the `intellisense` one bound to double-tab. To configure that, you have to set up the handler property and enable the double-tab functionality, which you can do by setting the `DoubleTabEnabled` configuration property to `$true`. Here is the command that gets double-tab functionality going:

```
PS> $PowerTabConfig.DoubleTabEnabled = $true
PS> $PowerTabConfig.DefaultHandler = 'ConsoleList'
PS> $PowerTabConfig.AlternateHandler = 'intellisense'
```

Again, remember to call `Export-tabExpansionConfig` if you like the setting and want it persisted for your future shell sessions.

The Tab Expansion Database

The tab expansion database is a local cache of all items that can be returned as command completions. It can contain many types of things: by default, it contains a list of .NET types, WMI classes, and much more. It can also contain custom entries that you can use as you like. The first convenience use that comes to mind is to use custom completion entries as code snippets. We can store a long string under a shorter key, type the key, and get the long portion back. More than one completion can be stored under the same key—if so, when we request completions for that key, we will get a list of options instead of only one item.

Adding custom tab expansions should be done through the `add-tabExpansion` function. It takes two parameters: the key and the expansion string. Let's now use that function to create a list of expansions serving as a favorites list. It will contain several folder paths, and we can use it to quickly navigate to any of them. Here is the code that adds two folders to the list:

```
PS> add-tabExpansion fav "c:\Windows\system32\WindowsPowerShell"
```

Filter	Text	Type
-----	----	----
fav	c:\Windows\system32\...	Custom

```
PS> add-tabExpansion fav "c:\Users\Hristo\Documents\WindowsPowerShell"
```

Filter	Text	Type
-----	----	----
fav	c:\Users\Hristo\Docu...	Custom

PowerTab has a special syntax for requesting custom completions: you have to type the name of the key followed by a caret symbol (^). To navigate quickly to a folder on our fav list,

type `cd fav^[tab]`, and select the item from the pop-up list. The result should look similar to the one shown in Figure 23-9.

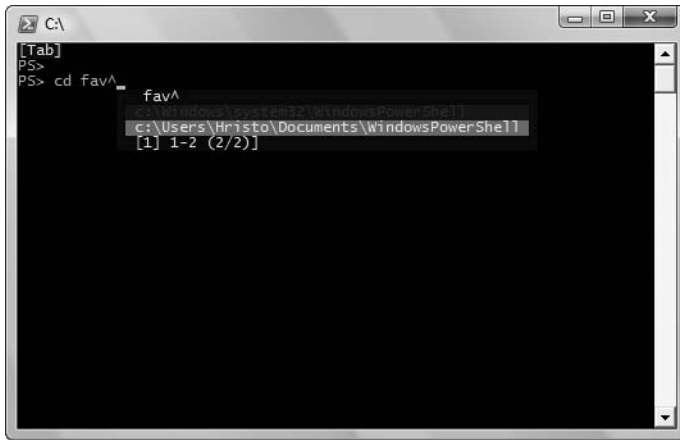


Figure 23-9. *PowerTab displaying custom completions.*

Just like when changing the `$PowerTabConfiguration` object, changes to the tab expansion database are not persisted across shell sessions. We need to explicitly save the modified tab expansion settings. To do that, we have to call the `Export-tabExpansionDataBase` function:

```
PS> Export-tabExpansionDataBase
Tabexpansion database exported to C:\Users\Hristo\Documents\WindowsPowerShell\TabExpansion.xml
PS>
```

Now, we can rest assured that our `fav` completion list will be available in all shell sessions.

Summary

PowerTab is a really great tool. Every day, I continue to be amazed at the sheer number of tab completion scenarios that it covers. The tool saves a lot of typing and can help you get productivity levels you have not dreamt were possible. I have noticed a strange thing happening to me—I use tab completions more and more as an exploration tool. It is not uncommon for me to spot classes and objects that I had not previously suspected even existed in a completion list, and thus discover a new cool feature that I can use in my work.

I bet that after half an hour of using PowerTab, you will never want to go back to the regular PowerShell tab expansion mechanism. Try it—you will not be sorry.

This chapter concludes our PowerShell adventure. We started by exploring the shell intricacies and went beyond simple commands to help you attain a profound understanding of how the shell components really work. We then went on to employ various programming techniques that use .NET, COM, and WMI objects to help us interact with the operating system and the network. Finally, I covered the most important, freely available, tools, which can immensely increase productivity. The techniques and tools described in the book are guaranteed to make you a better scripter. Use them!

Index

■ Numbers and symbols

- % alias, 124
- \$() block, 12
- @() block, 17
- <= indicator, 66
- => indicator, 66
- ! operator, 38–39
- < operator, 34, 60
- > operator, 34, 60, 170, 289
- >> operator, 34, 289
- += operator, 77
- \$_ variable, 59, 61, 77, 163
- `a (alert) symbol, 9
- & (ampersand) symbol, 70, 82
- * (asterisk) symbol, 13, 400
- ` (backtick) symbol, 9–10
- ^ (caret) symbol, 470
- `r (carriage return) symbol, 10
- { } (curly braces), 69
- \$ (dollar sign), 12, 64, 79
- . (dot) symbol, 13, 146, 148
- `f (form feed) symbol, 9
- `t (horizontal tab) symbol, 10
- `n (line feed) symbol, 10
- `0 (null) symbol, 9
- `f (page break) symbol, 9
- + (plus sign), 20, 71
- # (pound sign), 16
- ? (question mark), 57, 124
- 2> redirection operator, 170
- `v (vertical tab) symbol, 10

■ A

- \$a statement, 35
- absolute paths, 398, 402
- Accept-Encoding header, 322
- access control list (ACL), 63, 137, 245
- AccessToString property, 246
- Activator class, 368
- Active Directory
 - converting strings to entries, 30
 - deleting users from, 446
 - manipulating, 443–446
- add_Click() method, 86
- AddCode() method, 390
- Add-Content cmdlet, 136, 254, 288–289
- Add-EventHandler cmdlet, 458
- Add-Field function, 241
- Add-Member cmdlet, 238, 244
- Add-Method function, 242
- Add-Property function, 242
- Add-PSSnapIn cmdlet, 226–227, 450, 469
- add-tabExpansion function, 470
- administrator user privileges, 141
- advanced pager application, 446
- alert symbol (`a), 9
- alias properties, 6
- Alias provider, 109–110
- aliases. *See* command aliases
- AliasProperty, 239
- AllSigned policy level, 141, 211
- AlternateHandler configuration property, 470
- American Standard Code for Information Interchange (ASCII), 289
- AML. *See* Assistive Markup Language

- ampersand (&) symbol, 70, 82
- and operator, 37
- Application command type, 265
- Application event log, 303
- applications, shells embedded in, 219. *See also specific applications*
- \$args collection, 77, 142
- \$args variable, 73, 142–243
- ArgumentException errors, 175
- ArgumentNullException error, 175
- arrays, 16–21
 - Count property and, 6
 - creation syntax, 18
 - empty, 17
 - returning, 73
 - searching, 20–21
 - slices, 18
 - type conversions to, 28
- as parameter, 114
- AssemblyCache provider, 440–441
- Assistive Markup Language (AML), 258–262
- asString parameter, 63
- asterisk (*), 13, 400
- Asymmetric encryption algorithms, 198
- Atom feeds, 333–336
- audit logs, 63
- authentication, accessing resources
 - requiring, 324–328
- Authorization header, 327–328
- awk command, 56

B

- backspace (^b) , 9
- backtick (`) symbol, 9–10
- backward compatibility, 56
- band operator, 38–40
- base64 encryption, 327
- begin block, 76–77, 97
- begin parameter, 59
- BigEndianUnicode encoding, 292

- binary objects, turning into formatted text, 237
- binary operations, 38–40
- Bitmap objects, 436
- bitwise operations, 37–43
- blind copies, 353
- blocks, scope of, 83
- BOM sequences, 293
- Boolean conversions, 28, 41–42
- bor operator, 38–40
- bracket notation, 18
- branching instructions
 - if/else, 44–45
 - switch statement, 45–48
- break statement, 46, 51–53, 162–167
- breakpoints, simulating, 189–190
- built-in types, 8–24

- arrays, 16–21
- conversions, 21
- dictionaries, 21–24
- hash tables, 21–24
- numeric, 14–16
- strings, 8–14

- byte arrays, 290–291

C

- caret symbol (^), 470
- carriage return symbol (^r), 10
- case-insensitive operators, 36
- case-sensitive comparisons, 35–36
- casesensitive switch, 46
- cd command, 132–133, 421–422
- Cells property, 378
- Certificate provider, 128
- certificates
 - adding to Trusted Publishers store, 212
 - creating CA, 202–206
 - creating self-signed, 202–209
 - exporting, 209, 215
 - importing, 216

- issuing code-signing, 206–209
- managing, 199–202
- certification authorities (CAs), 198–199
- character arrays, converting strings to, 29
- character casing, 35
- checksums, 198
- *.chm files, 258
- CIM Query Language, 398
- CIM-XML protocol, 397
- class ID (CLSID), 368
- __CLASS property, 406
- classes
 - abstraction of, 240
 - adding members to all instances of, 244–247
 - See also specific classes*
- Clear-Content cmdlet, 136
- Clear-Host cmdlet, 222
- Clear-Item cmdlet, 134
- Clear-ItemProperty cmdlet, 136
- click() method, 387
- clike operator, 47
- clip.exe, 437
- clipboard helpers, 437–440
- CloseMainWindow method, 277
- cmatch operator, 47
- cmd.exe, 140
 - drive navigation and, 132–133
 - script execution and, 148–149
- CMD.EXE compatibility aliases, 119–121
- cmdlets, 56–58, 107. *See also specific cmdlets*
- code debugging. *See* debugging
- code signing, 197–199
- CodeObject property, 392
- CodePlex, 417, 449
- code-signing certificate, issuing, 206–209
- collections, 16–21
 - conditional expressions and, 43–44
 - Count property of, 6
 - executing actions on all objects in, 50–51
 - filtering, using Where-Object, 60
 - gathering statistics, 65
 - grouping, 64
 - processing, using ForEach-Object cmdlet, 59–60
 - searching, 20–21
 - sorting, 62–63
 - switch statement with, 47
- COM. *See* Component Object Model
- COM objects
 - built-in adapter for, 369
 - creating, 368–369
 - monikers for, 369
 - using, 368
 - WMI objects as, 395
 - wrappers, 368
- ComAdapter, 5
- command aliases, 107
 - built-in, 119–125
 - cmd.exe compatible, 119–121
 - complex, 117–118
 - convenience, 119, 123–124
 - creating, 108–109
 - deleting, 111
 - exporting and importing, 111–114
 - introduction to, 107
 - modifying, 110–111
 - name clashes, 115–117
 - persisting, for all shell sessions, 115
 - pitfalls of, 115
 - redefining default, 113
 - removing broken, 118–119
 - tips and techniques, 115–119
 - UNIX shell compatibility, 119–122
 - working with, 107–115
- command completion, 231–235, 465
- command tracing, 193–196
- CommandInfo object, 152

- commands
 - accessing information, 264–266
 - acting on objects (cmdlets), 56–59
 - chaining, 55, 66
 - combining simple, 66
 - dot-sourcing, 82–83
 - executing, 427–429
 - helper, 56
 - output from, 55
 - parsing problems, 56
 - PowerShell versus UNIX, 122
 - running from shell, 55
 - separating by function, 55
- CommandType property, 89
- commas, 73
- comma-separated files, exported objects to, 382
- Common Information Model (CIM), 396–397
- Compare-Object cmdlet, 58, 65–66
- comparison operators, 34–36
- compiled HTML help files, 258
- Component Object Model (COM)
 - automation interfaces with, 369–388
 - Internet Explorer, 384–388
 - Microsoft Excel, 378–383
 - Microsoft Word, 369–378
 - interfaces, 368
 - introduction to, 367
 - PowerShell support for, 368–369
 - workings of, 368
 - See also* COM objects
- Computer parameter, 301
- concatenation, 20
- conditional branches, 33
- conditional expressions, 33–44
 - Boolean conversions, 41–42
 - collections and, 43–44
 - defined, 33
 - implicit type conversions, 36–37
 - logic and bitwise operations, 37–40
 - string conversions, 42–43
 - value expressions, 34–36
- configuration code, saving in profile script, 223–225
- Connect-EventListener, 451
- connection options
 - debugging, 318–327
 - setting, 318–327
- console applications, 219
- console settings, working with saved, 226–228
- ConsoleList handler, 468
- consuming events, 449
- contains, 24
- Contains() method, 24
- ContainsKey() method, 24
- ContainsValue() method, 24
- content
 - encoding, 289–294
 - reading, 285–287
 - working with byte arrays, 290–291
 - writing, 288–289
- \$Contents variable, 316
- continue statement, 51–53, 161–162
- contracts, 368
- control flow, 33
- control structures, implementing new, 101–104
- Control-C event, 457
- convenience aliases, 119, 123–124
- conversion errors, 160–161
- Convert-Xml cmdlet, 433–434
- Copy-Item cmdlet, 135
- Copy-ItemProperty cmdlet, 137
- Count property, 6–7
- Count-Characters function, 183, 186–188
- counter data, 307–309
- counters, for loops, 50
- Create() method, 240–241

- CreateInstance() method, 368
- Create-Message function, 351–353
- Credential ability, 128
- Culture parameter, 63
- cURL, 316
- curly braces {}, 69
- currency, 16
- custom objects, creating, 240–244
- D**
- data
 - counter, 307–309
 - formats, 285
 - performance, 306
- data collections. *See* collections
- data extraction, from text, 294–296
- DataRowAdapter, 4
- DataRowViewAdapter, 5
- DCOM (Distributed COM), 367, 397
- Debug parameter, 157, 160, 183
- \$DebugPreference variable, 183, 185
- debugging, 177–196
 - command tracing, 193–196
 - generating diagnostic messages, 183–185
 - generating warnings, 186–187
 - print, 178–187
 - simulating breakpoints, 189–190
 - stepping through scripts, 188–190
 - tracing script execution details, 190–196
- decimal numbers, 15
- DefaultHandler property, 469
- Define-Class function, 240
- Definition property, 89, 266
- delegates, script blocks as, 85–86
- DependentServices property, 283
- __DERIVATION property, 406
- diagnostic messages, 297
 - color-coding, 178
 - generating, 183–185
 - verbose, 180–183
 - warnings, 186–187
- See also event logs*
- dictionaries, 21–24
- diff command, 58
- dir command, 93, 120, 444
- directory junctions, 422–423
- DirectoryEntryAdapter, 4
- DirectoryServices provider, 443–446
- Disconnect-EventListener cmdlet, 451
- Distributed COM (DCOM), 367, 397
- DivideByZeroException, 164
- DNS names, resolving, 426
- document object model (DOM)
 - programming interface, 387–388
- Document Paragraphs collection, 371
- Document.Content range, 373
- Documents collection, 370
- Documents.Add() method, 374
- Do-Events cmdlet, 459
- dollar sign (\$), 12, 64, 79
- dollar-sign expressions, 11
- domain controller, 425
- doskey.exe, 107
- dot (.) symbol, 13, 146
- dot notation, 2, 22
- dot-sourcing
 - commands, 82–83
 - script libraries and, 150
 - scripts, 146
- double negation, 42
- double quotes, 9
- DoubleTabEnabled configuration property, 470
- do-until loops, 49
- do-while loops, 49, 457
- DownloadFile method, 318, 345
- DownloadString method, 316
- drive providers, 134

- drives
 - creating new, 129
 - listing, 407
 - mapping network shares as local, 130
 - navigating to, 132–133
 - removing mapped, 131
 - scope of, 131–132
- dynamic item property providers, 137
- dynamic object generation, 392
- __DYNASTY property, 406
- E**
- EchoArgs program, 447
- Edit-File function, 418
- \$EDITOR global variable, 418
- EDSAC computer, 87
- elevate function, 429
- elevated privileges, 429
- else statement, 44, 45
- elseif statement, 45
- e-mail
 - attachments, 359–361
 - configuring recipients, 352–354
 - embedding media files in, 361–362
 - final script, 363–364
 - message headers, 355–357
 - message views, 357–359
 - multipart messages, 357–363
 - .NET framework support for, 349–350
 - read receipts, 356
 - reusable script for, 350–352
 - System.Net.Mail, 349–350
- embedded expressions, 12
- empty arrays, 17
- encoding
 - files, 289–294
 - Unicode, 291–294
 - UTF-8, 292
- Encoding parameter, 289–294
- encryption, 198–199
- end block, 76–77, 97
- end parameter, 59
- EnterNestedPrompt method, 189
- EntryWritten event, 452
- eq (equal) switch, 34, 46
- \$Error variable, 170–172
- error handling, 56
 - break statement for, 162–167
 - catching unknown or all, 163–168
 - common parameters, 157–160
 - error suppression, 161–162
 - introduction to, 155–157
 - interpreting error messages, 297
 - nonterminating errors, 170–172
 - parameters and, 74
 - parse errors, 169–170
 - passing up, 166
 - syntax errors, 169–170
 - trapping errors, 160–163
 - untrappable errors, 169–170
- error pipeline redirection, 170–171
- error reports, for scripts, 167–168
- error signaling, for scripts, 149
- ErrorAction parameter, 157–158
- ERRORLEVEL global variable, 149
- ErrorRecord, for nonterminating errors, 170
- errors
 - common, 447–448
 - ignoring, 102
 - raising, 172–177
 - nonterminating, 175–177
 - terminating, 172–175
 - See also* error handling
- ErrorVariable parameter, 157–160, 170
- eval function, 83
- Eval() method, 390–391
- event handling
 - example, 451–452
 - with PSEventing, 450–451
 - script blocks for, 457–459

- event logs, 297
 - accessing remote, 301–302
 - Application, 303
 - default, 298
 - event IDs, 300–301
 - listing, 298
 - security settings, 300
 - system, 452–454
 - Windows, 297–305
 - reading, 298–302
 - writing to, 302–305
- event queues, 450
- event source, 303
- eventhandler.ps1 script, 458
- EventLog class, 303
- events, 369, 449. *See also* PSEventing
- \$ExecutionContext variable, 84
- \$ExecutionContext.InvokeCommand object, 84
- Excel. *See* Microsoft Excel
- Excel.Application ProgID, 378
- Exception property, 160, 164
- exchangeable image file format (EXIF) metadata, 436
- Exclude ability, 129
- exclusive or operator (-xor), 38
- execution flow
 - branching instructions for, 44–48
 - loops, 48–53
- execution policy levels, 140–141
- exit statement, 147
- ExpandWildCards ability, 128
- Export-Alias cmdlet, 112, 228
- Export-Bitmap cmdlet, 436
- Export-CliXml cmdlet, 167
- Export-Console cmdlet, 227
- Export-Csv cmdlet, 382
- Export-tabExpansionDataBase function, 471

- expressions
 - evaluation of, 169
 - invoking strings as, 83–84
 - script blocks and, 71–75
- extended type system
 - configuration of, 245
 - external objects and, 239
 - object formatting, 247–251
 - PSObject type and, 237
- external objects, extending, 239
- external programs, working with, 316
- \$extraHeaders parameter, 357

F

- f string operator, 16
- Feed drive, 441
- Fiddler, 319–323
- file attributes, 39–40
- file compression, 424–425
- file delete error, 156
- file formats, 285
- file switch, 48
- file system abstraction, 444
- file system utilities, 421–425
 - compressing and archiving files, 424–425
 - navigation support, 421–422
 - NTFS helpers, 422–424
- FileInfo objects, 438
- files
 - archiving, 424–425
 - downloading from FTP server, 345–346
 - encoding, 289–294
 - finding string occurrences in, 295–296
 - reading content of, 285–287
 - saving downloaded, 317–318
 - uploading to FTP server, 346–347
 - working with byte arrays, 290–291
 - writing to, 288–289
- FileSystem provider, 127

- FileSystem.format.ps1xml file, 250
- FileSystemWatcher object, 449, 451–452
- Filter ability, 129
- filter keyword, 98
- FilterInfo object, 100
- filters, 97–101
- findstr command, 56
- first parameter, 61
- flag enumerations, 39, 43
- floating point numbers, 15
- flow control loops, 48–53
- for loops, 50, 101
- for operator, 59
- Force parameter, 16
- foreach cmdlet, 57, 331, 352
- foreach loops, 50–52
- ForEach-Object cmdlet, 57, 59–60, 75
- form feed symbol (^f), 9
- Format-* commands, 7
- Format-Custom cmdlet, 249
- Format-List cmdlet, 248, 388, 459
- Format-Table cmdlet, 248
- formatting cmdlets, 247–249
- formatting strings, 12
- Format-Wide cmdlet, 248
- Format-Xml cmdlet, 430–431
- FTP transactions, 345–347
- FullyQualifiedErrorId property, 160, 164
- function keyword, 87
- functional programming, 66–67
- FunctionInfo object, 100
- functions
 - calling, 88
 - concept of, 87
 - defining, 87–97
 - filter, 97–101
 - internals, 88–89
 - names, 87
 - nesting, 94
 - parameters, 89–92

- pipelines and, 97–101
- returning values, 93–94
- scope, 95
- scope rules, 94–97
- script blocks and, 101–106
- syntax for calling, 90
- See also specific functions*

G

- Gac drive, 440
- ge (greater-than-or-equal) switch, 35
- __GENUS property, 406
- GET requests, calling web services with, 338–340
- Get-Acl cmdlet, 137, 245
- Get-Alias cmdlet, 108–109, 193
- Get-Attribute() method, 106
- Get-AuthenticodeSignature, 212
- Get-ChildItem cmdlet, 135, 191–193
- Get-Clipboard cmdlet, 437–439
- Get-Command cmdlet, 88, 99, 109, 235, 264–266
- Get-Content cmdlet, 263, 285–287, 391
- Get-DhcpServer cmdlet, 425–426
- Get-DocumentText.ps1 script, 371
- Get-DocumentWords.ps1 script, 373
- Get-DomainController cmdlet, 425–426
- getElementById() method, 387
- getElementsByTagName() method, 388
- Get-Event cmdlet, 451–452, 457
- Get-EventBinding cmdlet, 451
- Get-EventLog cmdlet, 298–303
- Get-ExecutionPolicy cmdlet, 140
- Get-Help cmdlet, 253–256
 - advanced techniques, 259
 - parameter details, 256–257
- Get-HelpMatch function, 421
- Get-Host cmdlet, 220
- Get-Item cmdlet, 134
- Get-ItemProperty cmdlet, 136, 444
- Get-LibraryPath function, 153

- Get-Location cmdlet, 228
- Get-Member cmdlet, 7, 266–268
- Get-PfxCertificate cmdlet, 213
- Get-Process cmdlet, 34, 60, 64, 275–278, 383, 429
- Get-PscxAlias function, 419
- Get-PscxCmdlet, 419
- Get-PscxDrive cmdlet, 420
- Get-PscxVariable cmdlet, 420
- Get-PSDrive cmdlet, 129
- Get-PSProvider cmdlet, 127
- Get-PSSnapIn cmdlet, 226, 450
- GetResponse method, 329
- Get-ScriptDirectory function, 152
- Get-Service cmdlet, 281
- GetSpellingSuggestions() method, 377
- Get-TabExpansion cmdlet, 418
- GetType() method, 1–2
- GetTypeFromProgID() method, 368
- Get-Unique cmdlet, 63
- Get-Url cmdlet, 427
- Get-Variable cmdlet, 79–80, 91, 152, 183
- Get-WmiObject cmdlet, 301, 399–404, 410
- global error traps, 167–168
- global event queue, 450
- global feed store, Internet Explorer, 441–443
- global functions, for configuring shell, 228
- global scope, 80–81
- \$global variable, 81
- globally unique identifier (GUID), 368
- Google search, 271
- grep command, 56
- group, 58
- GroupBy property, 251
- GroupInfo objects, 64
- Group-Object cmdlet, 58, 64–65
- gt (greater-than) switch, 34
- GUID (globally unique identifier), 368

H

- hard links, 422–423
- hardware devices, querying, 407–408
- hash functions, 198
- hash tables, 21–24, 28
- Hashtable class, 28
- HEAD request, 328–329
- \$Headers parameter, 323
- help, using Internet to access, 268–273
- help command, 254–256, 446
- help system, 253–264
- helper commands, 56
- hexadecimal numbers, 15–16, 40
- horizontal tab symbol (``t`), 10
- \$host global variable, 220
- host object, 220
- \$host.UI property, 220
- \$host.UI.WriteLine() method, 221
- HotFixID property, 409
- HTTP Basic authentication, 327–328
- HTTP POST request, 384
- HTTP (Hypertext Transfer Protocol), 315
 - authentication mechanisms, 324–328
 - calling web services, with POST requests, 340–342
 - downloading files with, 316–318
 - GET requests, calling web services with, 338–340
 - HEAD request, 328–329
 - proxy servers and, 318–327
 - request headers, 322–323

I

- IConvertible .NET interface, 27
- IDictionary objects, 28
- if/else branches, 44, 45
- ignore function, 102
- image files, 435–437
- ImageMagick, 435
- images, viewing downloaded, 317

- ImplementingType property, 128
- implicit type conversions, 36–37
- Import-Alias cmdlet, 112–113, 228
- Import-Bitmap cmdlet, 435–436, 439
- Import-CliXml cmdlet, 168
- Include ability, 129
- includeEqual diff option, 279
- InnerException property, 161, 164
- Installer class, 450
- installutil command, 469
- \$instance value, 240
- Instrument-Function cmdlet, 182
- IntelliSense completion handler, 468–470
- Internet, accessing help on, 268–273
- Internet Explorer
 - automating, with COM, 384–388
 - extracting data from DOM tree, 387–388
 - scripting browser session, 384–387
 - global feed store, 441–443
 - protected mode, 385
- InternetExplorer.Application ProgID, 384
- InvocationInfo object, 160
- invoke operator (&), 70
- Invoke() method, 70, 85
- InvokeCommand property, 84
- Invoke-Expression cmdlet, 83–84, 89
- Invoke-Item cmdlet, 135, 317
- InvokeReturnAsIs method, 70
- ISA operator, 403
- IsFilter property, 101
- item container providers, 135–136
- item content providers, 136
- item property providers, 136
- item providers, 134–135
- item security descriptor providers, 137
- items, 127

J

- Join-Path cmdlet, 136, 153, 318

L

- language differences, in sorting, 63
- last parameter, 61
- le (less-than-or-equal-to) switch, 35
- LerchSnapIn snap-in, 468
- less pager, 446–447
- libraries
 - processes and, 279
 - problems with paths, 150–153
- Lib-User-DotSource.ps1, 150
- like operator, 42–43, 47
- line feed symbol (`n), 10
- LinkedResource objects, 361
- List cmdlet switch, 298
- Little Endian ordering, 292
- Live.com search, 272
- local scope, 81
- local variables, 82, 91
- logging, 63. *See also* event logs
- logic operations, 37–40
- logic operators
 - ! operator, 38–39
 - and operator, 37
 - not operator, 38–39
 - or operator, 37
 - xor operator, 38
- logical drives, 407
- LogName parameter, 299
- loop function, 102
- loops, 48–53
 - controlling execution of, 51–53
 - do-until, 49
 - do-while, 49
 - for, 50
 - foreach, 50–52
 - while, 48–49, 101
- lt (less-than) switch, 33–34

M

- machine-parsable output, 55
- macro recording, 378
- MailAttachment objects, 360
- MailMessage class, 349, 355
- MainModule property, 279
- MainWindowTitle property, 278
- makecert.exe tool, 202–209, 215
- MAML (Microsoft Assistive Markup Language), 258
- MamlCommandHelpInfo objects, 259
- ManagementEventWatcher class, 454–455
- ManagementObjectAdapter class, 3
- mandatory parameters, 75
- match operator, 13, 42–43, 47
- Matches() method, 294
- \$MaximumErrorCount variable, 171
- Measure-Object cmdlet, 58, 65
- media files, embedding in e-mail messages, 361–362
- member sets, 7
- member types, 238
- members, adding to all instances of class, 244–247
- MemberType parameter, 238
- memory leaks, detecting, 309–310
- message headers, 355–357
- message read receipts, 355
- message views, 357
- meta_class WMI class, 403–406
- methods, 1. *See also specific methods*
- Microsoft Assistive Markup Language (MAML), 258
- Microsoft Developer Network (MSDN) web site, 268–271
- Microsoft Excel, automating, 378–383
 - exiting, 380
 - modifying workbook content, 382–383
 - reading cell values from document, 378–381
- Microsoft Word, automating, 369–378
 - creating and modifying documents, 373–376
 - help resources, 377–378
 - opening documents and extracting text, 370–373
 - using spell checker, 376–377
 - viewing application window, 376
- Microsoft.PowerShell.Core snap-in, 128
- MIME (Multipurpose Internet Mail Extensions), 357
- Modify-Name function scope, 145
- Modules collection, 279
- modulo operator (%), 52
- Monadism, 66
- monikers, 369, 398
- mount command, 122
- Move-Item cmdlet, 136
- Move-ItemProperty cmdlet, 137
- MSScriptControl COM object, 389–392
- MSScriptControl.ScriptControl ProgID, 389
- multiple branches, switch statement and, 45–48
- Multipurpose Internet Mail Extensions (MIME), 357
- \$MyInvocation variable, 152

N

- \$name variable, 145
- named parameters, 73
- __NAMESPACE property, 406
- navigation providers, 136
- navigation support, 421–422
- ne (not-equal) switch, 34
- negative indexes, 19
- nested error traps, 165–166
- nested scopes, 84
- .NET 3.0, 367
- .NET framework, 1
 - class names, completing with PowerTab, 463
 - COM support in, 368

- e-mail support, 349–350
 - event handling in, 85, 450–451
 - HTTP protocol and, 315–316
 - object interoperability and, 367
 - See also specific classes*
 - .NET global assembly cache, reading from, 440–441
 - .NET objects
 - commands as, 56
 - control of, 449
 - creating, 369
 - strings pointing to, 30
 - wrappers, 380
 - .NET Remoting infrastructure, 367
 - network adapters, 411
 - network configuration, WMI and, 411
 - network drives, mapped, 407
 - network shares, mapping as local drives, 130
 - network utilities, 425–427
 - NetworkCredential object, 325, 350
 - new keyword, 241
 - New() constructor method, 241
 - New-Alias cmdlet, 108
 - Newest parameter, 299
 - New-HardLink cmdlet, 422
 - New-Item cmdlet, 135, 442
 - New-ItemProperty cmdlet, 137
 - New-Junction cmdlet, 423
 - New-Object cmdlet, 174, 369
 - New-PSDrive cmdlet, 122
 - news feeds, 332–337
 - NextValue() method, 307
 - None ability, 129
 - nonempty values, 41
 - nonterminating errors, 155–156, 170–172, 175–177
 - not operator, 38–39
 - note properties, 7, 241
 - Notepad++, 139
 - notepad.exe, 419
 - NoteProperty, 239
 - NTFS file system, 422–424
 - NTLM (NT LAN Manager) authentication, 324
 - \$null values, 42, 74
 - null symbol (`0), 9
 - numeric range notation, 17
 - numeric types, 14–16
 - decimal numbers, 15
 - floating-point numbers, 15
 - hexadecimal numbers, 15–16
 - as strings, 16
 - System.Int32, 14
- 0**
- object adapters, 3–5
 - object discovery, 397–398
 - object formatting, extending, 247–251
 - object members, 2
 - object methods
 - completing with PowerTab, 464
 - using, 2–3
 - See also specific methods*
 - object notation, 22
 - object paths, 402
 - object properties
 - accessing, 2
 - adding or deleting, 60–62
 - object queries, 398–400
 - object types
 - accessing, 1–2
 - aliases, 25
 - built-in, 8–24
 - arrays, 16–21
 - dictionaries, 21–24
 - hash tables, 21–24
 - numeric, 14–16
 - strings, 8–14
 - configuration, 245–247
 - defining properties for, 244–247

- implicit type conversions, 36–37
 - modifying, 237–247
 - type conversion, 26–30, 36–37
 - type extensions, 5–8, 239
 - type literals, 25–30, 402
 - object views, extended, 6
 - object-based pipelines, 56–59, 238
 - advantages of, 56
 - functional programming and, 66–67
 - objects
 - accessing information about, 266–268
 - adding members to single, 238–240
 - comparing, 65–66
 - creating new, 240–244
 - extending external, 239
 - grouping, 64–65
 - introduction to, 1
 - modifying, 237–247
 - paths, 398
 - piping, 238
 - sorting collections of, 62–63
 - storing, 63–64
 - views, 8
 - operating system (OS)
 - querying, with WMI, 408–411
 - updates, 408–409
 - operating-specific tracing, 193–196
 - operations, used in expressions, 71
 - or operator, 37
 - Out-Clipboard cmdlet, 439
 - out-dataGridView function, 467
 - Out-Default cmdlet, 94
 - Out-String cmdlet, 439
- P**
- page break symbol (␣), 9
 - param statement, 73–74, 142
 - parameter names, 73, 257
 - parameter parameter, 256–257
 - parameters
 - accessing by index, 73
 - converting, 74
 - default values, 75, 90
 - error handling, 157–160
 - extracting, for methods and property setters, 243
 - function, 89–92
 - mandatory, 75
 - missing, 74
 - named, 73
 - passing, 71–77, 142–447
 - types of, 74
 - parent scopes, 78
 - parentheses, 71, 77
 - parse errors, 169–170
 - Parse() method, 27, 30
 - parse-time optimizations, 169
 - partitions, 407
 - password-protected pages, accessing, 324–327
 - PATH environment variable, 140
 - Path parameter, 295
 - __PATH property, 406
 - path specifier, 140
 - Pattern parameter, 295
 - percentage format, 16
 - performance counters, 297, 306–313
 - performance data, 306
 - Performance Monitor Users group, 307–308
 - ping utility, 426
 - Ping-Host cmdlet, 426
 - pipeline trees, 63–64
 - pipelines
 - advantages of, 55
 - function, 97–101
 - object-based, 56–59, 238
 - processing input, 75–83
 - scope and, 83
 - text-based, 55–56

- plus sign (+) operator, 20, 71
- POST requests, calling web services with, 340–342
- pound sign (#), 16
- PowerShell
 - COM support in, 368–369
 - executing scripts from other environments, 148–149
 - flexible type system, 1–8
 - help system, 253–264
 - object orientation, 253
 - object-based pipelines in, 56–59
 - script code interoperability, 389–392
 - snap-ins, 226–228
 - type conversion in, 26–30
 - WMI support, 398–400
- PowerShell Community Extensions (PSCX), 410
 - AssemblyCache provider, 440–441
 - clipboard helpers, 437–440
 - DirectoryServices provider, 443–446
 - feed provider, 441–443
 - file system utilities, 421–425
 - compressing and archiving files, 424–425
 - navigation support, 421–422
 - NTFS helpers, 422–424
 - helper functions, 419–421
 - image files, 435–437
 - installation and configuration, 417–421
 - introduction to, 417
 - network utilities, 425–427
 - process and command execution, 427–429
 - tab expansion, 418
 - utility applications, 446–448
 - XML tools, 430–434
- PowerShell drives. *See* drives
- powershell.exe, 219–223
- PowerTab
 - configuring, 462, 467–468
 - custom completions, 471
 - data grid pop-up windows, 466–467
 - installation, 462–463
 - introduction to, 461
 - pressing Tab twice in, 470
 - tab completion handler, 468–470
 - tab expansion database, 470–471
 - workings of, 463–467
- PowerTabConfig.xml file, 462, 467–468
- PrependPath parameter, 246
- print debugging, 178–187
 - generating debug output, 183–185
 - generating warnings, 186–187
 - verbose output, 180–183
- PriorityClass property, 280
- Private Bytes performance counter, 309
- private keys, 198–199, 203
- private scope, 81
- process block, 76–77
- process modules, 279–280
- Process object, 7
- process parameter, 60
- process section, 97–99
- Process type, 6
- Process.Start() static method, 277
- processes
 - executing, 427–429
 - introduction to, 275
 - listing, 276
 - priority setting, 280–281
 - publisher information, 280
 - starting and stopping, 277–278
 - windows, 278–279
 - working with, 275–278
- profile scripts, 223–225, 419
- program ID string (ProgID), 368

- programs
 - automating with COM, 369–388
 - Internet Explorer, 384–388
 - Microsoft Excel, 378–383
 - Microsoft Word, 369–378
 - branching instructions, 44–48
 - installed, 409–410
 - monitoring, 309–313
 - detecting memory leaks, 309–310
 - hung programs, 310–312
 - unexpected exits, 312–313
 - prompt function, 228–231
 - prompt settings, changing, 228–231
 - properties
 - adding or deleting, 60–62
 - defining for object types, 244–247
 - __PROPERTY_COUNT property, 406
 - property sets, 247
 - property types, 239
 - PropertySet element, 246–247
 - provider capabilities, 128–129, 134–137
 - provider-qualified paths, 133–134
 - providers
 - drives for, 129–132, 134
 - dynamic item property, 137
 - enumerating, 127–129
 - item, 127, 134–135
 - item container, 135–136
 - item content, 136
 - item property, 136
 - item security descriptor, 137
 - navigation, 136
 - proxy servers
 - debugging, 319–320
 - working with, 318–327
 - .ps1 file extension, 139
 - PSAdapted view, 8
 - PSBase property, 3–4, 105
 - PSBase view, 8
 - PSConfiguration view, 7
 - PSCX. *See* PowerShell Community Extensions
 - \$PscxTextEditorPreference variable, 420
 - PSEventing snap-in, 226, 301
 - detecting script termination, 456–457
 - event-handling mechanics, 450–451
 - example, 451–452
 - handling WMI events, 454–456
 - introduction to, 449–450
 - monitoring system event logs, 452–454
 - using script blocks as event handlers, 457–459
 - PSExtended property, 6–7
 - PSExtended view, 8
 - PSHostRawUserInterface object, 221
 - PSHostUserInterface object, 220–221
 - PSInvalidCastException object, 161, 164
 - PSMemberSet object, 7
 - PSObject type, 3, 27, 237
 - adding properties and methods to, 238
 - creating instances of, 240–244
 - object operations and, 239
 - PSObject view, 8
 - PSParentPath property, 251
 - PSReference object, 92
 - PSResources view, 7
 - PSSnapIn class, 450
 - PSSnapInInstaller class, 450
 - PSTypeConverter class, 27
 - PSVariable object, 79
 - public key infrastructure (PKI), 199
 - public keys, 198–199
- ## Q
- query languages, 398
 - Quit() method, 380
 - quotation marks, 9

■ **R**

Raise-NonTerminatingError function, 156
Raise-TerminatingError function, 156–157
raising errors, 156–157, 172–177
Range objects, 370–375, 378
Range() method, 381
raw user interface object, 221
RecordNumber property, 302
redirection operator, 170
reference values, in functions, 92
-regex switch, 47
Regex.Replace() method, 14
regular expression matching, 13–14
regular expressions, 29, 56, 294–296
relative paths, 398, 402
Reliability and Performance Monitor, 306–307, 311
__RELPATH property, 406
remote events, event log for, 301–302
RemoteSigned policy level, 141
Remove() method, 23
Remove-Item cmdlet, 111, 135
Remove-ItemProperty cmdlet, 137
Remove-Junction cmdlet, 423
Remove-PSDrive cmdlet, 131
Remove-ReparsePoint cmdlet, 423–424
Rename-Item cmdlet, 135
Rename-ItemProperty cmdlet, 137
reparse points, 423
-replace operator, 14
request headers, 322–323
Resize-Bitmap cmdlet, 436
Resolve-Host cmdlet, 426
Resolve-Path cmdlet, 135
Resource Description Framework (RDF), 332
resources, authenticating, 324–328
Responding property, 278
Restart-Service cmdlet, 282
Restricted policy level, 140–141

Resume-Service cmdlet, 282
return statement, 72, 93, 147
rows, selecting, 61
RSS feeds, 332–337, 441–443
RSS file formats, 332
Run() method, 390–391
running objects table, 369
RuntimeException error, 164

■ **S**

scientific notation, 16
scope
 drive, 131–132
 functions and, 94–97
 global, 80
 parent, 78
 prefixes, 81–82
 variable, 77–83, 144–145
scope namespace prefixes, 96
-scope parameter, 80, 145
script blocks, 59, 87, 263
 advantages of, 69
 defining, 69–71
 as delegates, 85–86
 evaluating, 390–391
 as event handlers, 457–459
 executing, 70–71
 functions and, 67, 101–106
 invoking, 70
 in object pipeline, 66
 passing parameters, 71–77
 processing pipeline input, 75–83
 queries, 263
 returning values, 71–75
 as strategies, 104–106
 type conversions, 29
 variable scope and, 77–83
 variables, 69
 See also scripts
script control object, 389–392

- script global scope, 145
- script libraries
 - developing and maintaining, 149–153
 - dot-sourcing and, 150
 - library path problem, 150–153
- Script properties, 7
- script publisher, trusting a, 212
- script scope prefix, 81, 145
- ScriptBlock property, 89, 101
- ScriptMethod type, 239
- Scriptomatic tool, 412–414
- ScriptProperty, 239
- scripts
 - building reusable, 350–352
 - certifying origin of, 198–199
 - creating, 139–140
 - detecting termination, 456–457
 - dot-sourcing, 146
 - error signaling, 149
 - execution policy levels, 140–141
 - exiting, 147–149
 - global error traps in, 167–168
 - integrity of, 198
 - interoperability, 389–392
 - introduction to, 139
 - invoking, 140–141
 - passing parameters, 142–146
 - PowerShell, executing from other environments, 148–149
 - returning values, 146–147
 - running on other machines, 215–216
 - signing, 197–199, 211–215
 - simulating breakpoints, 189–190
 - stepping through, 188–190
 - tracing execution details, 190–196
 - variables in, 144–146
- search strategy, with script blocks, 105–106
- Search-Google function, 271
- Search-LiveCom function, 272
- Search-Msdn function, 270
- secure-by-default building principle, 140
- Security log, 300
- sed command, 56
- select, 57
- SELECT clause, 60
- Select-Object cmdlet, 57, 60–62
- Select-String cmdlet, 295–296
- Select-Xml cmdlet, 433
- self-signed certificates, 202–209, 215
- semicolons, 73
- Send-Message function, 351
- __SERVER property, 406
- services
 - administering, 281–283
 - analyzing dependencies, 283
 - configuring, 283
 - introduction to, 275
 - starting and stopping, 282
- ServicesDependedOn property, 283
- Set-Acl cmdlet, 137
- Set-Alias cmdlet, 110, 114
- Set-Attachments function, 360
- Set-AuthenticodeSignature cmdlet, 211
- Set-Clipboard cmdlet, 439
- Set-Content cmdlet, 111, 136, 139, 288–290, 317
- Set-ExecutionPolicy cmdlet, 141, 211
- Set-FileTime cmdlet, 419
- Set-Headers function, 355–356
- Set-Item cmdlet, 110, 135
- Set-ItemProperty cmdlet, 136
- Set-LinkedResources function, 361–362
- Set-Location cmdlet, 132–133
- Set-MultiPartBodies function, 358
- Set-PSDebug cmdlet, 188–190
- Set-Service cmdlet, 283
- Set-Variable cmdlet, 79–80, 91, 145
- shell environment

- command completion, 231–235
- configuring, 221–223
- primary function of, 55
- prompt settings, 228–231
- user profile scripts, 223–225
- working with snap-ins, 226–228
 - See also* PowerShell
- shell hosts, 219–223
- shell operations, tracing, 190–193
- ShouldProcess method, 129
- Show-MsdnHelp function, 271
- Simple Network Management Protocol (SNMP), 395
- Simple Object Access Protocol (SOAP),
 - calling web services with, 342–345
- single quotes, 9, 11
- snap-ins
 - installing, 450
 - working with, 226–228
 - See also specific snap-ins*
- software, querying with WMI, 408–411
- sort command, 57
- Sort-Object cmdlet, 57, 62–63
- SpellingSuggestion objects, 377
- Split-Path cmdlet, 152
- *SQL* wildcard, 281
- Start-KeyHandler cmdlet, 456
- Start-Process cmdlet, 277, 428–429
- Start-Service cmdlet, 282
- Start-Transcript cmdlet, 265
- static members, accessing, 30
- static switch parameter, 267
- statistics, gathering, 65
- stdout stream, 171
- Stop-KeyHandler cmdlet, 456
- Stop-Process cmdlet, 64, 275–278
- Stop-Service cmdlet, 282
- string conversions, 42–43
- string interpolation, 11–12
- string literals, 9–11
- string operations, 8–14
- string operators, productivity boosting,
 - 12–13
- String.Replace() method, 14
- strings, 8–14
 - comparing, 35–36
 - immutable nature of, 2
 - invoking as expressions, 83–84
 - numbers as, 16
 - replacing, 14
 - types conversions, 28–30
- subroutines, 87. *See also* functions
- substrings, replacing, 14
- __SUPERCLASS property, 406
- Suspend-Service cmdlet, 282
- switch statement, 45–48, 66
- switches, as comparison operators, 34
- symbolic links (symlinks), 423, 429
- syntax errors, 169–170
- system event logs, 452–454. *See also* event logs
- system monitoring
 - with performance counters, 306–313
 - program monitoring, 309–313
 - tools for, 297
 - with Windows event log, 297–305
- System namespace prefix, 25
- System.ArgumentException object, 174
- System.ArgumentNullException object, 175
- System.Collection.Hashtable object, 21
- System.Diagnostics.EventLog class, 303, 452
- System.Diagnostics.PerformanceCounter object, 307
- System.Diagnostics.Process object, 5
- System.EventHandler class, 85
- System.Globalization.NumberFormatInfo class, 16
- System.Int32 object, 14
- System.IO.FileAttributes enumeration, 39, 42
- System.Management.Automation class, 70

- System.Management.
 - Automation.ErrorRecord object, 160
- System.Management.Automation.
 - FunctionInfo object, 88
- System.Net.Mail class, 349–350, 363, 365
- System.Net.WebClient class, 320–321, 324–326, 345–347
- System.Net.WebRequest class, 328–329
- System.String object type, 8
- System.Web.HttpUtility .NET class, 270

T

- tab completions
 - handler for, 468–470
 - support for, 461
 - See also* PowerTab
- tab expansion, 418
- tab expansion database, 470–471
- TabExpansion function, 231–235
- table view definitions, 251
- TableControl property, 251
- Tar archive, 425
- TargetObject property, 160
- Tee-Object cmdlet, 58, 63–64
- terminating errors, 155–157, 172–175
- terminator usage, inadvertent, 447
- Test-Path cmdlet, 135, 179
- Test-Xml cmdlet, 431–432
- text
 - encoding, 289–294
 - extracting data from, 294–296
 - parsing between commands, 55–56
- text editors, 139
- text files, 285
- text formatting system, 237
- text-based pipelines, 55–56
- __THIS special property, 403
- \$this variable, 238–242
- thrashing, 276
- throw statement, 174

- TIFF images, 436
- ToString() method, 288
- \$total variable, 77
- trace function, 103
- Trace-Command cmdlet, 193–196
- traps, error, 160–168
 - generic, 164
 - global, 167–168
 - nested, 165–166
- Trusted Publishers store, 212, 216
- Trusted Root Authorities store, 215
- Trusted Sites zone, 385
- type aliases, 25
- type cast operators, 27
- type configuration, 245–247
- type conversions, 26
 - built-in rules, 27–30
 - implicit, 36–37
- type extensions, 5–8, 239, 247–251
- type literals, 25–30, 402
- type system, 1–8, 237
- TypeConverter class, 27
- types, modifying, 237–247
- types.ps1xml file, 244–247

U

- UI-related operations, 220
- Unicode Consortium, 294
- Unicode encodings, 291–294
- Unified Modeling Language (UML), 396
- Uniform Resource Locators (URLs), 316, 330–331
- UNIX shell compatibility aliases, 119–122
- unknown errors, 163–168
- Unrestricted policy level, 141
- untrappable errors, 169–170
- Update-FormatData cmdlet, 250
- Update-TypeData cmdlet, 245–247
- URLs, 316, 330–331
- user profile scripts, 223–225, 350–351

User-Agent header, 322, 323
UTF-8 BOM sequence, 292
UTF-8 encoding, 292
utility applications, 446–448

■ V

value comparisons, 34–36
-valueOnly parameter, 79
values, returning, 146–147
variable parameter, 64
variable scope, 77–83, 91
 functions and, 94–97
 prefix for denoting, 80
 setting, 144–145
variables
 assigning values to, 79
 converting to PSReference, 28
 defined, 1
 local, 82, 91
 modifying, 80
 passing to functions, 91–92
 scripts and, 144–146
 substitution, 11–12
VBA code, 383
VBScript code, 398, 412
verbose messages, 180–183
Verbose parameter, 157, 160
\$VerbosePreference variable, 180–181
VeriSign root certificate, 215
vertical tab symbol (^v), 10
view names, 251
views, extending, 249–251
ViewSelectedBy element, 251
Visual Basic for Applications (VBA)
 environment, 377, 383

■ W

WaitForLoad function, 386
\$WarningPreference variable, 186
warnings, generating, 186–187
Web. *See* World Wide Web

Web browsers, 316, 322
Web pages
 accessing password protected, 324–327
 checking for broken links, 330–332
 downloading, 316–318
 downloading content, 427
 verifying existence of, 328–329
Web Service Description Language
 (WSDL), 338
Web services, calling, 338–344
 with HTTP GET requests, 338–340
 with HTTP POST requests, 340–342
 with SOAP protocol, 342–345
Web Services for Management
 (WS-Management), 397
Web sites
 opening from within PowerShell, 268
 testing and validating, 328–332
Web-Based Enterprise Management
 (WBEM), 396–398
WebClient class, 316, 321, 324–326, 345–347
WebRequest class, 328–329
wget, 316
\$what parameter, 142
where cmdlet, 57, 436, 444, 457
Where-Object cmdlet, 21, 57, 299
while loops, 48–49, 101
-wildcard operator, 47
wildcards
 as file path parameter, 287
 matching, 13
 in parameters, 257
 searching for services using, 281
Win32_LogicalDisk class, 407
Win32_NetworkAdapter object, 411
Win32_NTEventLogFile object, 301
Win32_Process technique, 404
Win32_Product instances, 409
Windows Communications Foundation, 367

- Windows event log, 297–305
 - reading, 298–302
 - writing to, 302–305
 - Windows Event Viewer control panel, 298
 - Windows Installer, 409–410
 - Windows Management Instrumentation (WMI)
 - class names, 464, 466
 - event handling, with PSEventing, 454–456
 - generating code, 411–414
 - history of, 395–396
 - infrastructure, 301–302
 - introduction to, 395
 - listing classes, 404–406
 - network configuration queries, 411
 - PowerShell support for, 398–400
 - query tool, 399–400
 - querying hardware devices, 407–408
 - querying software, 408–411
 - referencing classes, 402–403
 - system properties, 406
 - WBEM and, 396–398
 - Windows regional settings, 381
 - Windows Remote Management (WinRM), 397
 - Windows Script Host code, interoperability of, 389–392
 - Windows-integrated authentication, 324
 - WMI. *See* Windows Management Instrumentation
 - WMI objects, 3, 395
 - classes of, 396
 - converting strings to, 29
 - converting WQL queries into, 401–402
 - event retrieval with, 301
 - language support for, 401–404
 - WMI Query Language (WQL), 301, 398
 - Word.Application ProgID, 369
 - Words collection, 373
 - Workbook objects, 378
 - Workbooks collection, 378
 - Worksheet object, 378
 - World Wide Web (WWW)
 - connecting to, through proxy server, 318–327
 - downloading files from, 316–318
 - See also* Web pages, Web sites
 - World Wide Web Consortium (W3C), 340
 - WQL (WMI Query Language), 301, 398
 - WQL queries, converting to WMI objects, 401
 - Write-BZip2, 425
 - Write-Clipboard cmdlet, 439–440
 - Write-Debug cmdlet, 183–185
 - WriteEntry() method, 304
 - Write-Error cmdlet, 175–177, 187, 303
 - WriteErrorException, 176
 - Write-Event function, 305
 - Write-GZip cmdlet, 425
 - Write-Host cmdlet, 94, 103, 108, 178–180, 221
 - Write-Verbose cmdlet, 180–183
 - Write-Warning cmdlet, 186–187, 303
 - Write-Zip cmdlet, 424
 - WSDL (Web Service Description Language), 338
- X**
- XML documents, converting, 433
 - XML namespaces, defining, 433
 - XML tools, 430–434
 - XML validation, 431–432
 - XML Web services. *See* Web services
 - XmlNodeAdapter, 5
 - xor (exclusive or) operator, 38
 - XPath, 433
 - X-Receiver header, 353
 - XSLT transformations, 433–434
- Z**
- ZIP archive format, 424