



Windows PowerShell Pocket Reference
by Lee Holmes

Publisher: O'Reilly
Pub Date: May 20, 2008
Print ISBN-13: 978-0-596-52178-3
Pages: 174

[Table of Contents](#)
[Index](#)

Overview

This portable reference to Windows PowerShell summarizes both the command shell and scripting language, and provides a concise reference to the major tasks that make PowerShell so successful. It's an ideal on-the-job tool for Windows administrators who don't have time to plow through huge books or search online. Written by Microsoft PowerShell team member Lee Holmes, and excerpted from his *Windows PowerShell Cookbook*, *Windows PowerShell Pocket Reference* offers up-to-date coverage of PowerShell's 1.0 release. You'll find information on .NET classes and legacy management tools that you need to manage your system, along with chapters on how to write scripts, manage errors, format output, and much more. Beginning with a whirlwind tour of Windows PowerShell, this convenient guide covers:

- PowerShell language and environment
- Regular expression reference
- PowerShell automatic variables
- Standard PowerShell verbs
- Selected .NET classes and their uses
- WMI reference
- Selected COM objects and their uses
- .NET string formatting
- .NET datetime formatting

An authoritative source of information about PowerShell since its earliest betas, Lee Holmes' vast experience lets him incorporate both the "how" and the "why" into the book's discussions. His relationship with the PowerShell and administration community -- through newsgroups, mailing lists, and his informative blog [Lee Holmes](#) -- gives him insight into problems faced by administrators and PowerShell users alike. If you're ready to learn this powerful tool without having to break stride in your routine, this is the book you want.





Windows PowerShell Pocket Reference
by Lee Holmes

Publisher: O'Reilly
Pub Date: May 20, 2008
Print ISBN-13: 978-0-596-52178-3
Pages: 174

Table of Contents
Index

Copyright
Preface

- Chapter 1. A Whirlwind Tour of Windows PowerShell
 - Section 1.1. Introduction
 - Section 1.2. An Interactive Shell
 - Section 1.3. Structured Commands (Cmdlets)
 - Section 1.4. Deep Integration of Objects
 - Section 1.5. Administrators As First-Class Users
 - Section 1.6. Composable Commands
 - Section 1.7. Techniques to Protect You from Yourself
 - Section 1.8. Common Discovery Commands
 - Section 1.9. Ubiquitous Scripting
 - Section 1.10. Ad-Hoc Development
 - Section 1.11. Bridging Technologies
 - Section 1.12. Namespace Navigation Through Providers
 - Section 1.13. Much, Much More
- Chapter 2. PowerShell Language and Environment
 - Section 2.1. Commands and Expressions
 - Section 2.2. Comments
 - Section 2.3. Variables
 - Section 2.4. Booleans
 - Section 2.5. Strings
 - Section 2.6. Numbers
 - Section 2.7. Arrays and Lists
 - Section 2.8. Hashtables (Associative Arrays)
 - Section 2.9. XML
 - Section 2.10. Simple Operators
 - Section 2.11. Comparison Operators
 - Section 2.12. Conditional Statements
 - Section 2.13. Looping Statements
 - Section 2.14. Working with the .NET Framework
 - Section 2.15. Writing Scripts, Reusing Functionality
 - Section 2.16. Managing Errors
 - Section 2.17. Formatting Output
 - Section 2.18. Capturing Output
 - Section 2.19. Tracing and Debugging
 - Section 2.20. Common Customization Points
- Chapter 3. Regular Expression Reference
- Chapter 4. PowerShell Automatic Variables
- Chapter 5. Standard PowerShell Verbs
- Chapter 6. Selected .NET Classes and Their Uses
- Chapter 7. WMI Reference
- Chapter 8. Selected COM Objects and Their Uses
- Chapter 9. .NET String Formatting
 - Section 9.1. String Formatting Syntax
 - Section 9.2. Standard Numeric Format Strings
 - Section 9.3. Custom Numeric Format Strings
- Chapter 10. .NET DateTime Formatting
 - Section 10.1. Custom DateTime Format Strings

Index



Copyright

Copyright © 2008, O'Reilly Media. All rights reserved.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The *Pocket Reference* series designations, *Windows PowerShell Pocket Reference*, the image of a box turtle, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.





Preface

Windows PowerShell introduces a revolution to the world of system management and command-line shells. From its object-based pipelines, to its administrator focus, to its enormous reach into other Microsoft management technologies, PowerShell drastically improves the productivity of administrators and power-users alike.

Much of this power comes from providing access to powerful technologies: an expressive scripting language, regular expressions, the .NET Framework, Windows Management Instrumentation (WMI), COM, the Windows registry, and much more.

Although help for these technologies is independently available, it is scattered, unfocused, and buried among documentation intended for a developer audience.

To solve that problem, this Pocket Reference summarizes the Windows PowerShell command shell and scripting language, while also providing a concise reference for the major tasks that make it so successful.

P.1. Font Conventions

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, filenames, and file extensions.

`Constant width`

Indicates computer coding in a broad sense. This includes commands, options, elements, variables, attributes, keys, requests, functions, methods, types, classes, modules, properties, parameters, values, objects, events, event handlers, XML and XHTML tags, macros, and keywords.

`Constant width bold`

Indicates commands or other text that the user should type literally.

`Constant width italic`

Indicates text that should be replaced with user-supplied values or values determined by context.

P.2. Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

<http://www.oreilly.com/catalog/9780596521783>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

<http://www.oreilly.com>

P.3. Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at <http://safari.oreilly.com>.





Chapter 1. A Whirlwind Tour of Windows PowerShell

Introduction

An Interactive Shell

Structured Commands (Cmdlets)

Deep Integration of Objects

Administrators As First-Class Users

Composable Commands

Techniques to Protect You from Yourself

Common Discovery Commands

Ubiquitous Scripting

Ad-Hoc Development

Bridging Technologies

Namespace Navigation Through Providers

Much, Much More

1.1. Introduction

When learning a new technology, it is natural to feel bewildered at first by all the unfamiliar features and functionality. This perhaps rings especially true for users new to Windows PowerShell, because it may be their first experience with a fully featured command-line shell. Or worse, they've heard stories of PowerShell's fantastic integrated scripting capabilities and fear being forced into a world of programming that they've actively avoided until now.

Fortunately, these fears are entirely misguided: PowerShell is a shell that both grows with you and grows on you. Let's take a tour to see what it is capable of:

- PowerShell works with standard Windows commands and applications. You don't have to throw away what you already know and use.
- PowerShell introduces a powerful new type of command. PowerShell commands (called *cmdlets*) share a common *Verb-Noun* syntax and offer many usability improvements over standard commands.
- PowerShell understands objects. Working directly with richly structured objects makes working with (and combining) PowerShell commands immensely easier than working in the plain-text world of traditional shells.
- PowerShell caters to administrators. Even with all its advances, PowerShell focuses strongly on its use as

an *interactive shell*, and the experience of entering commands in a running PowerShell application.

- PowerShell supports discovery. Using three simple commands, you can learn and discover almost anything PowerShell has to offer.
- PowerShell enables ubiquitous scripting. With a fully fledged scripting language that works directly from the command line, PowerShell lets you automate tasks with ease.
- PowerShell bridges many technologies. By letting you work with .NET, COM, WMI, XML, and Active Directory, PowerShell makes working with these previously isolated technologies easier than ever before.
- PowerShell simplifies management of data stores. Through its provider model, PowerShell lets you manage data stores using the same techniques you already use to manage files and folders.

We'll explore each of these attributes in this introductory tour of PowerShell.





Chapter 1. A Whirlwind Tour of Windows PowerShell

Introduction

An Interactive Shell

Structured Commands (Cmdlets)

Deep Integration of Objects

Administrators As First-Class Users

Composable Commands

Techniques to Protect You from Yourself

Common Discovery Commands

Ubiquitous Scripting

Ad-Hoc Development

Bridging Technologies

Namespace Navigation Through Providers

Much, Much More

1.1. Introduction

When learning a new technology, it is natural to feel bewildered at first by all the unfamiliar features and functionality. This perhaps rings especially true for users new to Windows PowerShell, because it may be their first experience with a fully featured command-line shell. Or worse, they've heard stories of PowerShell's fantastic integrated scripting capabilities and fear being forced into a world of programming that they've actively avoided until now.

Fortunately, these fears are entirely misguided: PowerShell is a shell that both grows with you and grows on you. Let's take a tour to see what it is capable of:

- PowerShell works with standard Windows commands and applications. You don't have to throw away what you already know and use.
- PowerShell introduces a powerful new type of command. PowerShell commands (called *cmdlets*) share a common *Verb-Noun* syntax and offer many usability improvements over standard commands.
- PowerShell understands objects. Working directly with richly structured objects makes working with (and combining) PowerShell commands immensely easier than working in the plain-text world of traditional shells.
- PowerShell caters to administrators. Even with all its advances, PowerShell focuses strongly on its use as

an *interactive shell*, and the experience of entering commands in a running PowerShell application.

- PowerShell supports discovery. Using three simple commands, you can learn and discover almost anything PowerShell has to offer.
- PowerShell enables ubiquitous scripting. With a fully fledged scripting language that works directly from the command line, PowerShell lets you automate tasks with ease.
- PowerShell bridges many technologies. By letting you work with .NET, COM, WMI, XML, and Active Directory, PowerShell makes working with these previously isolated technologies easier than ever before.
- PowerShell simplifies management of data stores. Through its provider model, PowerShell lets you manage data stores using the same techniques you already use to manage files and folders.

We'll explore each of these attributes in this introductory tour of PowerShell.



1.2. An Interactive Shell

At its core, PowerShell is first and foremost an interactive shell. While it supports scripting and other powerful features, its focus as a shell underpins everything.

Getting started in PowerShell is a simple matter of launching *PowerShell.exe* rather than *cmd.exe*—the shells begin to diverge as you explore the intermediate and advanced functionality, but you can be productive in PowerShell immediately.

To launch Windows PowerShell, click:

Start → All Programs → Windows PowerShell 1.0 → Windows PowerShell

Or alternatively, click:

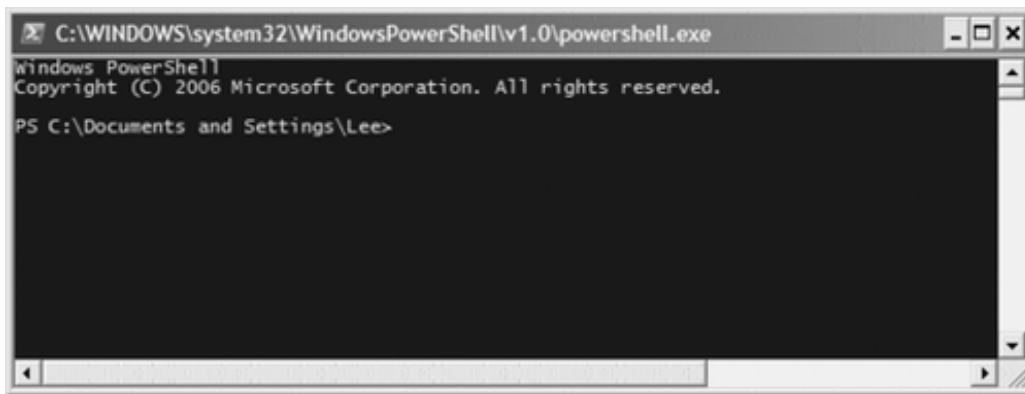
Start → Run

and then type:

`PowerShell`

A PowerShell prompt window opens that's nearly identical to the traditional command prompt window of Windows XP, Windows Server 2003, and their many ancestors. The `PS>` prompt indicates that PowerShell is ready for input, as shown in [Figure 1-1](#).

Figure 1-1. Windows PowerShell, ready for input



Once you've launched your PowerShell prompt, you can enter DOS-style and Unix-style commands for navigating around the filesystem, just as you would with any Windows or Unix command prompt—as in the interactive session shown in [Example 1-1](#).

[Example 1-1](#). Entering standard DOS-style file manipulation commands in response to the PowerShell prompt produces the same results you get when you use them with any other Windows shell

```

Code View:
PS C:\Documents and Settings\Lee> function Prompt { "PS >" }
PS >pushd .
PS >cd \
PS >dir

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

Mode                LastWriteTime         Length Name
----                -
d-----          11/2/2006   4:36 AM          $WINDOWS.~BT
d-----          5/8/2007   8:37 PM          Blurpark
d-----         11/29/2006   2:47 PM          Boot
d-----         11/28/2006   2:10 PM          DECHECK
d-----         10/7/2006   4:30 PM          Documents and Settings
d-----          5/21/2007   6:02 PM          F&SC-demo
d-----          4/2/2007   7:21 PM          Inetpub
d-----          5/20/2007   4:59 PM          Program Files
d-----          5/21/2007   7:26 PM          temp
d-----          5/21/2007   8:55 PM          Windows
-a----          1/7/2006  10:37 PM             0 autoexec.bat
-ar-s         11/29/2006   1:39 PM          8192 BOOTSECT.BAK
-a----          1/7/2006  10:37 PM             0 config.sys
-a----          5/1/2007   8:43 PM        33057 RUU.log
-a----          4/2/2007   7:46 PM          2487 secedit.INTEG.RAW

PS >popd
PS >pwd

Path
----
C:\Documents and Settings\Lee

```

As shown in [Example 1-1](#), you can use the `pushd`, `cd`, `dir`, `pwd`, and `popd` commands to store the current location, navigate around the filesystem, list items in the current directory, and then return to your original location. Try it!



The `pushd` command is an alternative name (alias) to the much more descriptively named PowerShell command, `Push-Location`. Likewise, the `cd`, `dir`, `popd`, and `pwd` commands all have more memorable counterparts.

Although navigating around the filesystem is helpful, so is running the tools you know and love, such as `ipconfig` and `notepad`. Type the command name and you'll see results like those shown in [Example 1-2](#).

Example 1-2. Windows tools and applications, such as `ipconfig`, run in PowerShell just as they do in the Windows prompt

```
PS >ipconfig

Windows IP Configuration

Ethernet adapter Wireless Network Connection 4:

    Connection-specific DNS Suffix  . : hsd1.wa.comcast.net.
    IP Address. . . . . : 192.168.1.100
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.1.1
PS >notepad
(notepad launches)
```

Entering `ipconfig` displays the IP addresses of your current network connections. Entering `notepad` runs—as you'd expect—the Notepad editor that ships with Windows. Try them both on your own machine.



1.3. Structured Commands (Cmdlets)

In addition to supporting traditional Windows executables, PowerShell introduces a powerful new type of command called a *cmdlet* (pronounced *command-let*.) All cmdlets are named in a *Verb-Noun* pattern, such as `Get-Process`, `Get-Content`, and `Stop-Process`.

```
PS >Get-Process -Name lsass
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
668	13	6228	1660	46		932	lsass

In this example, you provide a value to the `ProcessName` parameter to get a specific process by name.



Once you know the handful of common verbs in PowerShell, learning how to work with new nouns becomes much easier. While you may never have worked with a certain object before (such as a Service), the standard `Get`, `Set`, `Start`, and `Stop` actions still apply. For a list of these common verbs, see [Chapter 5](#).

You don't always have to type these full cmdlet names, however. PowerShell lets you use the `Tab` key to auto-complete cmdlet names and parameter names:

```
PS >Get-Pr<Tab> -N<Tab> lsass
```

For quick interactive use, even that may be too much typing. For improved efficiency, PowerShell defines aliases for all common commands and lets you define your own. In addition to alias names, PowerShell only requires that you type enough of the parameter name to disambiguate it from the other parameters in that cmdlet. PowerShell is also case-insensitive. Using the built-in `gps` alias that represents the `Get-Process` cmdlet (along with parameter shortening), you can instead type:

```
PS >gps -n lsass
```

Going even further, PowerShell supports *positional parameters* on cmdlets. Positional parameters let you provide parameter values in a certain position on the command line, rather than having to specify them by name. The `Get-Process` cmdlet takes a process name as its first positional parameter. This parameter even supports wildcards:

```
PS >gps l*s
```



1.4. Deep Integration of Objects

PowerShell begins to flex more of its muscle as you explore the way it handles structured data and richly functional objects. For example, the following command generates a simple text string. Since nothing captures that output, PowerShell displays it to you:

```
PS >"Hello World"  
Hello World
```

The string you just generated is, in fact, a fully functional object from the .NET Framework. For example, you can access its `Length` property, which tells you how many characters are in the string. To access a property, you place a dot between the object and its property name:

```
PS >"Hello World".Length  
11
```

All PowerShell commands that produce output generate that output as objects as well. For example, the `Get-Process` cmdlet generates a `System.Diagnostics.Process` object, which you can store in a variable. In PowerShell, variable names start with a `$` character. If you have an instance of Notepad running, the following command stores a reference to it:

```
$process = Get-Process notepad
```

Since this is a fully functional `Process` object from the .NET Framework, you can call methods on that object to perform actions on it. This command calls the `Kill()` method, which stops a process. To access a method, you place a dot between the object and its method name:

```
$process.Kill()
```

PowerShell supports this functionality more directly through the `Stop-Process` cmdlet, but this example demonstrates an important point about your ability to interact with these rich objects.





1.5. Administrators As First-Class Users

While PowerShell's support for objects from the .NET Framework quickens the pulse of most users, PowerShell continues to focus strongly on administrative tasks. For example, PowerShell supports `MB` (for megabyte) and `GB` (for gigabyte) as some of the standard administrative constants. For example, how many disks will it take to back up a 40 GB hard drive to CD-ROM?

```
PS >40GB / 650MB  
63.0153846153846
```

Just because PowerShell is an administrator-focused shell doesn't mean you can't still use the .NET Framework for administrative tasks, though! In fact, PowerShell makes a great calendar. For example, is 2008 a leap year? PowerShell can tell you:

```
PS >[DateTime]::IsLeapYear(2008)  
True
```

Going further, how might you determine how much time remains until summer? The following command converts "06/21/2008" (the start of summer) to a date, and then subtracts the current date from that. It stores the result in the `$result` variable, and then accesses the `TotalDays` property.

```
PS >$result = [DateTime] "06/21/2008" - [DateTime]::Now  
PS >$result.TotalDays  
283.0549285662616
```





1.6. Composable Commands

Whenever a command generates output, you can use a *pipeline character* (|) to pass that output directly to another command. If the second command understands the objects produced by the first command, it can operate on the results.

You can chain together many commands this way, creating powerful compositions out of a few simple operations. For example, the following command gets all items in the `Path1` directory and moves them to the `Path2` directory:

```
Get-Item Path1\* | Move-Item -Destination Path2
```

You can create even more complex commands by adding additional cmdlets to the pipeline. In [Example 1-3](#), the first command gets all processes running on the system. It passes those to the `Where-Object` cmdlet, which runs a comparison against each incoming item. In this case, the comparison is `$_ .Handles -ge 500`, which checks whether the `Handles` property of the current object (represented by the `$_` variable) is greater than or equal to 500. For each object in which this comparison holds true, you pass the results to the `Sort-Object` cmdlet, asking it to sort items by their `Handles` property. Finally, you pass the objects to the `Format-Table` cmdlet to generate a table that contains the `Handles`, `Name`, and `Description` of the process.

[Example 1-3](#). You can build more complex PowerShell commands by using pipelines to link cmdlets, as shown in this example with `Get-Process`, `Where-Object`, `Sort-Object`, and `Format-Table`

```
PS >Get-Process |
>>   Where-Object { $_.Handles -ge 500 } |
>>   Sort-Object Handles |
>>   Format-Table Handles,Name,Description -Auto
>>
```

Handles	Name	Description
588	winlogon	
592	svchost	
667	lsass	
725	csrss	
742	System	
964	WINWORD	Microsoft Office Word
1112	OUTLOOK	Microsoft Office Outlook
2063	svchost	





1.7. Techniques to Protect You from Yourself

While aliases, wildcards, and composable pipelines are powerful, their use in commands that modify system information can easily be nerve-wracking. After all, what does this command do? Think about it, but don't try it just yet:

```
PS >gps [b-t]*[c-r] | Stop-Process
```

It appears to stop all processes that begin with the letters `b` through `t` and end with the letters `c` through `r`. How can you be sure? Let PowerShell tell you. For commands that modify data, PowerShell supports `-WhatIf` and `-Confirm` parameters that let you see what a command *would* do:

```
PS >gps [b-t]*[c-r] | Stop-Process -whatif
What if: Performing operation "Stop-Process" on Target
"ctfmon (812)".
What if: Performing operation "Stop-Process" on Target
"Ditto (1916)".
What if: Performing operation "Stop-Process" on Target
"dsamain (316)".
What if: Performing operation "Stop-Process" on Target
"ehrecvr (1832)".
What if: Performing operation "Stop-Process" on Target
"ehSched (1852)".
What if: Performing operation "Stop-Process" on Target
"EXCEL (2092)".
What if: Performing operation "Stop-Process" on Target
"explorer (1900)".
(...)
```

In this interaction, using the `-WhatIf` parameter with the `Stop-Process` pipelined command lets you preview which processes on your system will be stopped before you actually carry out the operation.

Note that this example is not a dare! In the words of one reviewer:

Not only did it stop everything, but on Vista, it forced a shutdown with only one minute warning!

It was very funny though.... At least I had enough time to save everything first!





1.8. Common Discovery Commands

While reading through a guided tour is helpful, I find that most learning happens in an ad-hoc fashion. To find all commands that match a given wildcard, use the `Get-Command` cmdlet. For example, by entering the following, you can find out which PowerShell commands (and Windows applications) contain the word `process`.

```
PS >Get-Command *process*
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Get-Process	Get-Process [[-Name] <Str...
Application	qprocess.exe	c:\windows\system32\qproc...
Cmdlet	Stop-Process	Stop-Process [-Id] <Int32...

To see what a command such as `Get-Process` does, use the `Get-Help` cmdlet, like this:

```
PS >Get-Help Get-Process
```

Since PowerShell lets you work with objects from the .NET Framework, it provides the `Get-Member` cmdlet to retrieve information about the properties and methods that an object, such as a .NET `System.String`, supports. Piping a string to the `Get-Member` command displays its type name and its members:

Code View:

```
PS >"Hello World" | Get-Member
```

```
TypeName: System.String
```

Name	MemberType	Definition
----	-----	-----
(...)		
PadLeft	Method	System.String PadLeft(Int32 tota...
PadRight	Method	System.String PadRight(Int32 tot...
Remove	Method	System.String Remove(Int32 start...
Replace	Method	System.String Replace(Char oldCh...
Split	Method	System.String[] Split(Params Cha...
StartsWith	Method	System.Boolean StartsWith(String...
Substring	Method	System.String Substring(Int32 st...
ToChar-		System.Char[] ToCharArray(), Sys...
ArrayMethod		
ToLower	Method	System.String ToLower(), System...
ToLower-	Method	System.String ToLowerInvariant()
Invariant		
ToString	Method	System.String ToString(), System...
ToUpper	Method	System.String ToUpper(), System...
ToUpper-	Method	System.String ToUpperInvariant()
Invariant		
Trim	Method	System.String Trim(Params Char[]...
TrimEnd	Method	System.String TrimEnd(Params Cha...
TrimStart	Method	System.String TrimStart(Params C...
Chars	Parameter-	System.Char Chars(Int32 index) {...
	izedProperty	
Length	Property	System.Int32 Length {get;}



1.9. Ubiquitous Scripting

PowerShell makes no distinction between the commands you type at the command line and the commands you write in a script. This means that your favorite cmdlets work in scripts and that your favorite scripting techniques (such as the `foreach` statement) work directly on the command line.

For example, to add up the handle count for all running processes:

```
PS >$handleCount = 0
PS >foreach($process in Get-Process) { $handleCount +=
    $process.Handles }
PS >$handleCount
19403
```

While PowerShell provides a command (`Measure-Object`) to measure statistics about collections, this short example shows how PowerShell lets you apply techniques that normally require a separate scripting or programming language.

In addition to using PowerShell scripting keywords, you can also create and work directly with objects from the .NET Framework. PowerShell becomes almost like the C# immediate mode in Visual Studio. In [Example 1-4](#), you see how PowerShell lets you easily interact with the .NET Framework.

Example 1-4. Using objects from the .NET Framework to retrieve a web page and process its content

```
PS >$webClient = New-Object System.Net.WebClient
PS >$content = $webClient.DownloadString("http://blogs.msdn.com/
PowerShell/rss.aspx")
PS >$content.Substring(0,1000)
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet type="text/xsl" href="http://blogs.msdn.com/
utility/FeedStylesheets/rss.xsl" media="screen"?>
<rss version="2.0"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:slash="http://purl.org/rss/1.0/modules/slash/"
xmlns:wfw="http://wellformedweb.org/CommentAPI/"><channel>
<title>Windo
(...)
```





1.10. Ad-Hoc Development

By blurring the lines between interactive administration and writing scripts, the history buffer of PowerShell sessions quickly becomes the basis for ad-hoc script development. In this example, you call the `Get-History` cmdlet to retrieve the history of your session. For each of those items, you get its `CommandLine` property (the thing you typed) and send the output to a new script file.

```
PS >Get-History | Foreach-Object { $_.CommandLine } >
    c:\temp\script.ps1
PS >notepad c:\temp\script.ps1
(save the content you want to keep)
PS >c:\temp\script.ps1
```



If this is the first time you've run a script in PowerShell, you will need to configure your Execution Policy. For more information, type `'help about_signing'`.





1.11. Bridging Technologies

We've seen how PowerShell lets you fully leverage the .NET Framework in your tasks, but its support for common technologies stretches even further. As [Example 1-5](#) shows, PowerShell supports XML.

Example 1-5. Working with XML content in PowerShell

```
PS >$xmlContent = [xml] $content
PS >$xmlContent

xml                xml-stylesheet    rss
---                -
                    rss

PS >$xmlContent.rss

version : 2.0
dc      : http://purl.org/dc/elements/1.1/
slash  : http://purl.org/rss/1.0/modules/slash/
wfw    : http://wellformedweb.org/CommentAPI/
channel : channel

PS >$xmlContent.rss.channel.item | select Title

title
---
CMD.exe compatibility
Time Stamping Log Files
Microsoft Compute Cluster now has a PowerShell Provider and
Cmdlets
The Virtuous Cycle: .NET Developers using PowerShell
(...)
```

And Windows Management Instrumentation (WMI):

```
PS >Get-WmiObject Win32_Bios

SMBIOSBIOSVersion : ASUS A7N8X Deluxe ACPI BIOS Rev 1009
Manufacturer      : Phoenix Technologies, LTD
Name              : Phoenix - AwardBIOS v6.00PG
SerialNumber      : xxxxxxxxxxxx
Version           : Nvidia - 42302e31
```

Or, as [Example 1-6](#) shows, Active Directory Service Interfaces (ADSI).

Example 1-6. Working with Active Directory in PowerShell

Code View:

```
PS >[ADSI] "WinNT://./Administrator" | Format-List *
```

```
UserFlags           : {66113}
MaxStorage          : {-1}
PasswordAge         : {19550795}
PasswordExpired     : {0}
LoginHours          : {255 255 255 255 255 255 255 255
                    255 255 255 255 255 255 255 255
                    255 255 255 255 255}
FullName            : {}
Description          : {Built-in account for
                    administering the computer/
                    domain}
BadPasswordAttempts : {0}
LastLogin           : {5/21/2007 3:00:00 AM}
HomeDirectory       : {}
LoginScript         : {}
Profile             : {}
HomeDirDrive        : {}
Parameters          : {}
PrimaryGroupID      : {513}
Name                : {Administrator}

MinPasswordLength   : {0}
MaxPasswordAge      : {3710851}
MinPasswordAge      : {0}
PasswordHistoryLength : {0}
AutoUnlockInterval  : {1800}
LockoutObservationInterval : {1800}
MaxBadPasswordsAllowed : {0}
RasPermissions      : {1}
objectSid           : {1 5 0 0 0 0 0 5 21 0 0 0 121 227
                    252 83 122 130 50 34 67 23 10 50
                    244 1 0 0}
```

Or, as Example 1-7 shows, even scripting traditional COM objects.

Example 1-7. Working with COM objects in PowerShell

```
PS >$firewall = New-Object -com HNetCfg.FwMgr
PS >$firewall.LocalPolicy.CurrentProfile
```

```
Type : 1
FirewallEnabled : True
ExceptionsNotAllowed : False
NotificationsDisabled : False
UnicastResponsesToMulti-
castBroadcastDisabled : False
RemoteAdminSettings : System._ _ComObject
IcmpSettings : System._ _ComObject
GloballyOpenPorts : {Media Center Extender Service,
Remote Media Center Experience,
Adam Test Instance, QWAVE...}
Services : {File and Printer Sharing, UPnP
Framework, Remote Desktop}
AuthorizedApplications : {Remote Assistance, Windows
Messenger, Media Center,
Trillian...}
```





1.12. Namespace Navigation Through Providers

Another avenue PowerShell provides for working with the system is a *providers*. PowerShell providers let you navigate and manage data stores using the same techniques you already use to work with the filesystem, as illustrated in [Example 1-8](#).

Example 1-8. Navigating the filesystem

```
PS >Set-Location c:\
PS >dir

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

Mode                LastWriteTime         Length Name
----                -
d-----          11/2/2006   4:36 AM          $WINDOWS.~BT
d-----          5/8/2007   8:37 PM          Blurpark
d-----         11/29/2006   2:47 PM          Boot
d-----         11/28/2006   2:10 PM          DECCKECK
d-----          10/7/2006   4:30 PM          Documents and
                        Settings
d-----          5/21/2007   6:02 PM          F&SC-demo
d-----          4/2/2007   7:21 PM          Inetpub
d-----          5/20/2007   4:59 PM          Program Files
d-----          5/21/2007  11:47 PM          temp
d-----          5/21/2007   8:55 PM          Windows
-a----          1/7/2006  10:37 PM             0 autoexec.bat
-ar-s         11/29/2006   1:39 PM          8192 BOOTSECT.BAK
-a----          1/7/2006  10:37 PM             0 config.sys
-a----          5/1/2007   8:43 PM        33057 RUU.log
-a----          4/2/2007   7:46 PM          2487 secedit.INTEG.RAW
```

This also works on the registry, as shown in [Example 1-9](#).

Example 1-9. Navigating the registry

```

PS >Set-Location HKCU:\Software\Microsoft\Windows\
PS >Get-ChildItem

    Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\
        Software\Microsoft\Windows

SKC  VC Name          Property
----  -
30   1 CurrentVersion   {ISC}
3    1 Shell            {BagMRU Size}
4    2 ShellNoRoam     {(default), BagMRU Size}

PS >Set-Location CurrentVersion\Run
PS >Get-ItemProperty .

(...)
FolderShare      : "C:\Program Files\FolderShare\
                  FolderShare.exe" /background
TaskSwitchXP     : d:\lee\tools\TaskSwitchXP.exe
ctfmon.exe       : C:\WINDOWS\system32\ctfmon.exe
Ditto            : C:\Program Files\Ditto\Ditto.exe
(...)

```

Or even the machine's certificate store, as Example 1-10 illustrates.

Example 1-10. Navigating the certificate store

```

PS >Set-Location cert:\CurrentUser\Root
PS >Get-ChildItem

    Directory: Microsoft.PowerShell.Security\Certificate::
        CurrentUser\Root

Thumbprint          Subject
-----
CDD4EEAE6000AC7F40C3802C171E30148030C072 CN=Microsoft Root
Certificate...
BE36A4562FB2EE05DBB3D32323ADF445084ED656 CN=Thawte
Timestamping CA,
OU...
A43489159A520F0D93D032CCAF37E7FE20A8B419 CN=Microsoft Root
Authority, ...
9FE47B4D05D46E8066BAB1D1BFC9E48F1DBE6B26 CN=PowerShell Local
Certifica...
7F88CD7223F3C813818C994614A89C99FA3B5247 CN=Microsoft
Authenticode(tm)...
245C97DF7514E7CF2DF8BE72AE957B9E04741E85 OU=Copyright (c)
1997 Microso...

(...)

```





1.13. Much, Much More

As exciting as this guided tour was, it barely scratches the surface of how you can use PowerShell to improve your productivity and systems management skills. For more information about getting started in PowerShell, see the "Getting Started" and "User Guide" files included in the Windows PowerShell section of your Start menu. For a cookbook-style guide to PowerShell (and hard-won solutions to its most common problems), you may be interested in the source of the material in this pocket reference: my book *Windows PowerShell Cookbook* (O'Reilly).





Chapter 2. PowerShell Language and Environment

Commands and Expressions

Comments

Variables

Booleans

Strings

Numbers

Arrays and Lists

Hashtables (Associative Arrays)

XML

Simple Operators

Comparison Operators

Conditional Statements

Looping Statements

Working with the .NET Framework

Writing Scripts, Reusing Functionality

Managing Errors

Formatting Output

Capturing Output

Tracing and Debugging

Common Customization Points

2.1. Commands and Expressions

PowerShell breaks any line that you enter into its individual units (*tokens*), and then interprets each token in one of two ways: as a command or as an expression. The difference is subtle: expressions support logic and flow control statements (such as `if`, `foreach`, and `throw`) while commands do not. You will often want to control the way that Windows PowerShell interprets your statements, so [Table 2-1](#) lists the available options.

Table 2-1. Windows PowerShell evaluation controls

Statement	Example	Explanation
Precedence control: ()	<pre>PS >5 * (1 + 2) 15 PS >(dir).Count 2276</pre>	Forces the evaluation of a command or expression, similar to how parentheses force the order of evaluation in a math expression.
Expression subparse: \$()	<pre>PS >"The answer is (2+2)" The answer is (2+2) PS >"The answer is \$(2+2)" The answer is 4 PS >\$value = 10 PS >\$result = \$(>> if(\$value -gt 0) { \$true } else { \$false } >>) >> PS >\$result True</pre>	Forces the evaluation of a command or expression, similar to how parentheses force the order of evaluation in a mathematical expression. However, a subparse is as powerful as a subprogram, and is required only when it contains logic or flow control statements. This statement is also used to expand dynamic information inside a string.
List evaluation: @()	<pre>PS >"Hello".Length 5 PS >@"Hello".Length 1 PS >(Get-ChildItem).Count 12 PS >(Get-ChildItem *.txt).Count PS >@(Get-ChildItem *.txt).Count 1</pre>	Forces an expression to be evaluated as a list. If it is already a list, it will remain a list. If it is not, PowerShell temporarily treats it as one.





Chapter 2. PowerShell Language and Environment

Commands and Expressions

Comments

Variables

Booleans

Strings

Numbers

Arrays and Lists

Hashtables (Associative Arrays)

XML

Simple Operators

Comparison Operators

Conditional Statements

Looping Statements

Working with the .NET Framework

Writing Scripts, Reusing Functionality

Managing Errors

Formatting Output

Capturing Output

Tracing and Debugging

Common Customization Points

2.1. Commands and Expressions

PowerShell breaks any line that you enter into its individual units (*tokens*), and then interprets each token in one of two ways: as a command or as an expression. The difference is subtle: expressions support logic and flow control statements (such as `if`, `foreach`, and `throw`) while commands do not. You will often want to control the way that Windows PowerShell interprets your statements, so [Table 2-1](#) lists the available options.

Table 2-1. Windows PowerShell evaluation controls

Statement	Example	Explanation
Precedence control: ()	<pre>PS >5 * (1 + 2) 15 PS >(dir).Count 2276</pre>	Forces the evaluation of a command or expression, similar to how parentheses force the order of evaluation in a math expression.
Expression subparse: \$()	<pre>PS >"The answer is (2+2)" The answer is (2+2) PS >"The answer is \$(2+2)" The answer is 4 PS >\$value = 10 PS >\$result = \$(>> if(\$value -gt 0) { \$true } else { \$false } >>) >> PS >\$result True</pre>	Forces the evaluation of a command or expression, similar to how parentheses force the order of evaluation in a mathematical expression. However, a subparse is as powerful as a subprogram, and is required only when it contains logic or flow control statements. This statement is also used to expand dynamic information inside a string.
List evaluation: @()	<pre>PS >"Hello".Length 5 PS >@"Hello".Length 1 PS >(Get-ChildItem).Count 12 PS >(Get-ChildItem *.txt).Count PS >@(Get-ChildItem *.txt).Count 1</pre>	Forces an expression to be evaluated as a list. If it is already a list, it will remain a list. If it is not, PowerShell temporarily treats it as one.





2.2. Comments

To create single-line comments, begin a line with the `#` character. Windows PowerShell does not support multiline comments, but you can deactivate larger regions of your script by placing them in a *here string*.

```
# This is a regular comment

# Start of the here string
$null = @"
function MyTest
{
    "This should not be considered a function"
}

$myVariable = 10;
"@
# End of the here string

# This is regular script again
```



See "Strings" to learn more about here strings.





2.3. Variables

Windows PowerShell provides several ways to define and access variables, as summarized in [Table 2-2](#).

Table 2-2. Windows PowerShell variable syntaxes

Syntax	Meaning
<code>\$simpleVariable = "Value"</code>	A simple variable name. The variable name must consist of alphanumeric characters. Variable names are not case sensitive.
<code>\${arbitrary!@###{var` }iable} = "Value"</code>	An arbitrary variable name. The variable name must be surrounded by curly braces, but may contain any characters. Curly braces in the variable name must be escaped with a backtick (`).
<code> \${c:\filename.extension}</code>	Variable "Get and Set Content" syntax. This is similar to the arbitrary variable name syntax. If the name corresponds to a valid PowerShell path, you can get and set the content of the item at that location by reading and writing to the variable.
<code>[datatype] \$variable = "Value"</code>	Strongly typed variable. Ensures that the variable may contain only data of the type you declare. PowerShell throws an error if it cannot coerce the data to this type when you assign it.
<code>\$SCOPE:variable</code>	Gets or sets the variable at that specific scope. Valid scope names are <code>global</code> (to make a variable available to the entire shell), <code>script</code> (to make a variable available only to the current script), <code>local</code> (to make a variable available only to the current scope and subscopes), and <code>private</code> (to make a variable available only to the current scope). The default scope is the <i>current</i> scope: <code>global</code> when defined interactively in the shell, <code>script</code> when defined outside any functions or script blocks in a script, and <code>local</code> elsewhere.
<code>New-Item Variable:\variable-Value value</code>	Creates a new variable using the Variable Provider.
<code>Get-Item Variable:\variableGet-Variable variable</code>	Gets the variable using the Variable Provider or <code>Get-Variable</code> cmdlet. This lets you access extra information about the variable, such as its options and description.
<code>New-Variable variable-Option option-Value value</code>	Creates a variable using the <code>New-Variable</code> cmdlet. This lets you provide extra information about the variable, such as its options and description.



Unlike some languages, PowerShell rounds (not truncates) numbers when it converts them to the `[int]` data type:

```
PS >(3/2)
1.5
PS >[int] (3/2)
2
```

To have PowerShell truncate a number, use the static `Truncate` method in the `Math` class:

```
PS >[Math]::Truncate(3/2)
1
```





2.4. Booleans

Boolean (true or false) variables are most commonly initialized to their literal values of `$true` and `$false`. When it evaluates variables as part of a Boolean expression (for example, an `if` statement), though, PowerShell maps them to a suitable Boolean representation, as listed in [Table 2-3](#).

Table 2-3. Windows PowerShell Boolean interpretations

Result	Boolean representation
<code>\$true</code>	True
<code>\$false</code>	False
<code>\$null</code>	False
Nonzero number	True
Zero	False
Nonempty string	True
Empty string	False
Nonempty array	True
Empty array	False
Hashtable (either empty or not)	True



2.5. Strings

Windows PowerShell offers several facilities for working with plain-text data.

2.5.1. Literal and Expanding Strings

To define a literal string (one in which no variable or escape expansion occurs), enclose it in single quotes:

```
$myString = 'hello 't $ENV:SystemRoot'
```

`$myString` gets the actual value of `hello 't $ENV:SystemRoot`.

To define an expanding string (one in which variable and escape expansion occurs), enclose it in double quotes:

```
$myString = "hello 't $ENV:SystemRoot"
```

`$myString` gets a value similar to `hello C:\WINDOWS`.

To include a single quote in a single-quoted string or a double quote in a double-quoted string, you may include two of the quote characters in a row:

```
PS >"Hello ""There""!"
Hello "There"!
PS >'Hello ''There''!'
Hello 'There'!
```



To include a complex expression inside an expanding string, use a subexpression. For example:

```
$prompt = "$(Get-Location) >"
```

`$prompt` gets a value similar to `c:\temp >`.

Accessing the properties of an object requires a subexpression:

```
$output = "Current script name is:
    $($myInvocation.MyCommand.Path)"
```

`$output` gets a value similar to `Current script name is c:\Test-Script.ps1`.

2.5.2. Here Strings

To define a *here string* (one that may span multiple lines), place the two characters `@` at the beginning, and the two characters `"@"` on their own line at the end.

For example:

```
$myHereString = @"
This text may span multiple lines, and may
contain "quotes".
"@
```

Here strings may be of either the literal (single-quoted) or expanding (double quoted) variety.

2.5.3. Escape Sequences

Windows PowerShell supports escape sequences inside strings, as listed in [Table 2-4](#).

Table 2-4. Windows PowerShell escape sequences

Sequence	Meaning
<code>`0</code>	The <i>null</i> character. Often used as a record separator.
<code>`a</code>	The <i>alarm</i> character. Generates a beep when displayed on the console.
<code>`b</code>	The <i>backspace</i> character. The previous character remains in the string but is overwritten when displayed on the console.
<code>`f</code>	A <i>form feed</i> . Creates a page break when printed on most printers.
<code>`n</code>	A <i>newline</i> .
<code>`r</code>	A <i>carriage return</i> . Newlines in PowerShell are indicated entirely by the <code>`n</code> character, so this is rarely required.
<code>`t</code>	A <i>tab</i> .
<code>`v</code>	A <i>vertical tab</i> .
<code>"</code> (Two single quotes)	A <i>single quote</i> , when in a literal string.
<code>"</code> "(Two double quotes)	A <i>double quote</i> , when in an expanding string.
<code>'<any other character></code>	That character, taken literally.





2.6. Numbers

PowerShell offers several options for interacting with numbers and numeric data.

2.6.1. Simple Assignment

To define a variable that holds numeric data, simply assign it as you would other variables. PowerShell automatically stores your data in a format that is sufficient to accurately hold it.

```
$myInt = 10
```

`$myInt` gets the value of `10`, as a (32-bit) integer.

```
$myDouble = 3.14
```

`$myDouble` gets the value of `3.14`, as a (53-bit, 9 bits of precision) double.

To explicitly assign a number as a long (64-bit) integer or decimal (96-bit, 96 bits of precision), use the long and decimal suffixes:

```
$myLong = 2147483648L
```

`$myLong` gets the value of `2147483648`, as a long integer.

```
$myDecimal = 0.999D
```

`$myDecimal` gets the value of `0.999`.

PowerShell also supports scientific notation:

```
$myPi = 3141592653e-9
```

`$myPi` gets the value of `3.141592653`.

The data types in PowerShell (integer, long integer, double, and decimal) are built on the .NET data types of the same name.

2.6.2. Administrative Numeric Constants

Since computer administrators rarely get the chance to work with numbers in even powers of ten, PowerShell offers the numeric constants of `gb`, `mb`, and `kb` to represent gigabytes, megabytes, and kilobytes, respectively:

```
PS >$downloadTime = (1gb + 250mb) / 120kb
PS >$downloadTime
10871.4666666667
```

2.6.3. Hexadecimal and Other Number Bases

To directly enter a hexadecimal number, use the hexadecimal prefix 0x:

```
$myErrorCode = 0xFE4A
```

`$myErrorCode` gets the integer value 65098.

The PowerShell scripting language does not natively support other number bases, but its support for interaction with the .NET Framework enables conversion to and from binary, octal, decimal, and hexadecimal:

```
$myBinary = [Convert]::ToInt32("101101010101", 2)
```

`$myBinary` gets the integer value of 2901.

```
$myOctal = [Convert]::ToInt32("1234567", 8)
```

`$myOctal` gets the integer value of 342391.

```
$myHexString = [Convert]::ToString(65098, 16)
```

`$myHexString` gets the string value of fe4a.

```
$myBinaryString = [Convert]::ToString(12345, 2)
```

`$myBinaryString` gets the string value of 11000000111001.



See "Working with the .NET Framework," later in this chapter, to learn more about using PowerShell to interact with the .NET Framework.



2.7. Arrays and Lists

2.7.1. Array Definitions

PowerShell arrays hold lists of data. The `@()` (*array cast*) syntax tells PowerShell to treat the contents between the parentheses as an array. To create an empty array, type:

```
$myArray = @()
```

To define a nonempty array, use a comma to separate its elements:

```
$mySimpleArray = 1, "Two", 3.14
```

Arrays may optionally be only a single element long:

```
$myList = , "Hello"
```

Or, alternatively (using the array cast syntax):

```
$myList = @("Hello")
```

Elements of an array do not need to be all of the same data type, unless you declare it as a strongly typed array. In the following example, the outer square brackets define a strongly typed variable (as mentioned in "Variables," earlier in this chapter), and `int[]` represents an array of integers:

```
[int[]] $myArray = 1, 2, 3.14
```

In this mode, PowerShell throws an error if it cannot convert any of the elements in your list to the required data type. In this case, it rounds `3.14` to the integer value of `3`.

```
PS >$myArray[2]  
3
```



To ensure that PowerShell treats collections of uncertain length (such as history lists or directory listings) as a list, use the list evaluation syntax `@(...)` described in "Commands and Expressions," earlier in this chapter.

Arrays can also be multidimensional "jagged" arrays—arrays within arrays:

```
$multiDimensional = @(  
    (1, 2, 3, 4),  
    (5, 6, 7, 8)  
)
```

`$multiDimensional[0][1]` returns `2`, coming from row 0, column 1.

`$multiDimensional[1][3]` returns 8, coming from row 1, column 3.

To define a multidimensional array that is not jagged, create a multidimensional instance of the .NET type. For integers, that would be an array of `System.Int32`:

```
$multidimensional = New-Object "Int32[,] " 2,4  
$multidimensional[0,1] = 2  
$multidimensional[1,3] = 8
```

2.7.2. Array Access

To access a specific element in an array, use the `[]` operator. PowerShell numbers your array elements starting at 0. Using `$myArray = 1,2,3,4,5,6` as an example:

```
$myArray[0]
```

Returns 1, the first element in the array.

```
$myArray[2]
```

Returns 3, the third element in the array.

```
$myArray[-1]
```

Returns 6, the last element in the array.

```
$myArray[-2]
```

Returns 5, the second-to-last element in the array.

You can also access ranges of elements in your array:

```
PS >$myArray[0..2]  
1  
2  
3
```

Returns elements 0 through 2, inclusive.

```
PS >$myArray[-1..2]  
6  
1  
2  
3
```

Returns the final element, wraps around, and returns elements 0 through 2, inclusive. PowerShell wraps around because the first number in the range is positive, and the second number in the range is negative.

```
PS >$myArray[-1..-3]  
6  
5
```

Returns the last element in the array through to the third-to-last element in the array in decreasing order. PowerShell does not wrap around (and therefore scans backward in this case) because both numbers in the range share the same sign.

2.7.3. Array Slicing

You can combine several of the above statements at once to extract more complex ranges from an array. Use the + sign to separate array ranges from explicit indexes:

```
$myArray[0,2,4]
```

Returns the elements at indices 0, 2, and 4.

```
$myArray[0,2+4..5]
```

Returns the elements at indices 0, 2, and 4 through 5, inclusive.

```
$myArray[,0+2..3+0,0]
```

Returns the elements at indices 0, 2 through 3 inclusive, 0, and 0 again.



You can use the array slicing syntax to create arrays as well:

```
$myArray = ,0+2..3+0,0
```





2.8. Hashtables (Associative Arrays)

2.8.1. Hashtable Definitions

PowerShell *hashtables* (or *associative arrays*) let you associate keys with values. To define a hashtable, use the syntax:

```
$myHashtable = @{ }
```

You can initialize a hashtable with its key/value pairs when you create it. PowerShell assumes that the keys are strings, but the values may be any data type.

```
$myHashtable = @{ Key1 = "Value1";  
"Key 2" = 1,2,3; 3.14 = "Pi" }
```

2.8.2. Hashtable Access

To access or modify a specific element in an associative array, you may use either the array-access or property-access syntax:

```
$myHashtable["Key1"]
```

Returns "Value1".

```
$myHashtable."Key 2"
```

Returns the array 1,2,3.

```
$myHashtable["New Item"] = 5
```

Adds "New Item" to the hashtable.

```
$myHashtable."New Item" = 5
```

Also adds "New Item" to the hashtable.





2.9. XML

PowerShell supports XML as a native data type. To create an XML variable, cast a string to the `[xml]` type:

```
$myXml = [xml] @"
<AddressBook>
  <Person contactType="Personal">
    <Name>Lee</Name>
    <Phone type="home">555-1212</Phone>
    <Phone type="work">555-1213</Phone>
  </Person>
  <Person contactType="Business">
    <Name>Ariel</Name>
    <Phone>555-1234</Phone>
  </Person>
</AddressBook>
"@
```

PowerShell exposes all child nodes and attributes as properties. When it does this, PowerShell automatically groups children that share the same node type:

```
$myXml.AddressBook
```

Returns an object that contains a `Person` property.

```
$myXml.AddressBook.Person
```

Returns a list of `Person` nodes. Each `Person` node exposes `contactType`, `Name`, and `Phone` as properties.

```
$myXml.AddressBook.Person[0]
```

Returns the first `Person` node.

```
$myXml.AddressBook.Person[0].ContactType
```

Returns `Personal` as the contact type of the first `Person` node.



The XML data type wraps the .NET [XmlDocument](#) and [XmlElement](#) classes. Unlike most PowerShell .NET wrappers, this wrapper does not expose the properties from the underlying class because they may conflict with the dynamic properties that PowerShell adds for node names.

To access properties of the underlying class, use the [PsBase](#) property. For example:

```
$myXml.PsBase.InnerXml
```

See "Working with the .NET Framework," later in this chapter, to learn more about using PowerShell to interact with the .NET Framework.



2.10. Simple Operators

Once you've defined your data, the next step is to work with it.

2.10.1. Arithmetic Operators

The arithmetic operators let you perform mathematical operations on your data, as shown in [Table 2-5](#).

Table 2-5. Windows PowerShell arithmetic operators

Operator	Meaning
+	<p>The <i>addition operator</i>. <code>\$leftValue + \$rightValue</code></p> <p>When used with numbers, returns their sum.</p> <p>When used with strings, returns a new string created by appending the second string to the first.</p> <p>When used with arrays, returns a new array created by appending the second array to the first.</p> <p>When used with hashtables, returns a new hashtable created by merging the two hashtables. Since hashtable keys must be unique, PowerShell returns an error if the second hashtable includes any keys already defined in the first hashtable.</p> <p>When used with any other type, PowerShell uses that type's addition operator (<code>op_Addition</code>) if it implements one.</p>
-	<p>The <i>subtraction operator</i>. <code>\$leftValue - \$rightValue</code></p> <p>When used with numbers, returns their difference.</p> <p>This operator does not apply to strings.</p> <p>This operator does not apply to arrays.</p> <p>This operator does not apply to hashtables.</p> <p>When used with any other type, PowerShell uses that type's subtraction operator (<code>op_Subtraction</code>) if it implements one.</p>
*	<p>The <i>multiplication operator</i>. <code>\$leftValue * \$rightValue</code></p> <p>When used with numbers, returns their product.</p> <p>When used with strings ("<code>=</code>" * <code>80</code>), returns a new string created by appending the string to itself the number of times you specify.</p>

Operator	Meaning
	<p>When used with arrays (<code>1..3 * 7</code>), returns a new array created by appending the array to itself the number of times you specify.</p> <p>This operator does not apply to hashtables.</p> <p>When used with any other type, PowerShell uses that type's multiplication operator (<code>op_Multiply</code>) if it implements one.</p>
/	<p>The <i>division operator</i>: <code>\$leftValue / \$rightValue</code></p> <p>When used with numbers, returns their quotient.</p> <p>This operator does not apply to strings.</p> <p>This operator does not apply to arrays.</p> <p>This operator does not apply to hashtables.</p> <p>When used with any other type, PowerShell uses that type's multiplication operator (<code>op_Division</code>) if it implements one.</p>
%	<p>The <i>modulus operator</i>: <code>\$leftValue % \$rightValue</code></p> <p>When used with numbers, returns the remainder of their division.</p> <p>This operator does not apply to strings.</p> <p>This operator does not apply to arrays.</p> <p>This operator does not apply to hashtables.</p> <p>When used with any other type, PowerShell uses that type's multiplication operator (<code>op_Modulus</code>) if it implements one.</p>
+= -= *= /= %=	<p><i>Assignment operators</i>: <code>\$variable operator = value</code></p> <p>These operators match the simple arithmetic operators (+, -, *, /, and %) but store the result in the variable on the lefthand side of the operator. It is a short form for <code>\$variable = \$variable operator value</code></p>



The `System.Math` class in the .NET Framework offers many powerful operations in addition to the native operators supported by PowerShell:

```
PS >[Math]::Pow([Math]::E, [Math]::Pi)
23.1406926327793
```

See "Working with the .NET Framework," later in this chapter, to learn more about using PowerShell to interact with the .NET Framework.

2.10.2. Logical Operators

The logical operators let you compare Boolean values, as shown in [Table 2-6](#).

Table 2-6. Windows PowerShell logical operators

Operator	Meaning
<code>-and</code>	<p><i>Logical AND.</i> <code>\$leftValue -and \$rightValue</code></p> <p>Returns <code>\$true</code> if both lefthand and righthand arguments evaluate to <code>\$true</code>. Returns <code>\$false</code> otherwise.</p> <p>You can combine several <code>-and</code> operators in the same expression:</p> <pre><code>\$value1 -and \$value2 -and \$value3 ...</code></pre> <p>PowerShell implements the <code>-and</code> operator as a short-circuit operator, and evaluates arguments only if all arguments preceding it evaluate to <code>\$true</code>.</p>
<code>-or</code>	<p><i>Logical OR.</i> <code>\$leftValue -or \$rightValue</code></p> <p>Returns <code>\$true</code> if the lefthand or righthand arguments evaluate to <code>\$true</code>. Returns <code>\$false</code> otherwise.</p> <p>You can combine several <code>-or</code> operators in the same expression:</p> <pre><code>\$value1 -or \$value2 -or \$value3 ...</code></pre> <p>PowerShell implements the <code>-or</code> operator as a short-circuit operator and evaluates arguments only if all arguments preceding it evaluate to <code>\$false</code>.</p>
<code>-xor</code>	<p><i>Logical Exclusive OR.</i> <code>\$leftValue -xor \$rightValue</code></p> <p>Returns <code>\$true</code> if either the lefthand or righthand argument evaluates to <code>\$true</code>, but not if both do. Returns <code>\$false</code> otherwise.</p>
<code>-not!</code>	<p><i>Logical NOT.</i> <code>-not \$value</code></p>

Operator	Meaning
	Returns <code>\$true</code> if its (only) righthand argument evaluates to <code>\$false</code> . Returns <code>\$false</code> otherwise.

2.10.3. Binary Operators

The binary operators listed in [Table 2-7](#) let you apply the Boolean logical operators bit by bit to the operator's arguments. When comparing bits, a 1 represents `$true`, while a 0 represents `$false`.

Table 2-7. Windows PowerShell binary operators

Operator	Meaning
<code>-band</code>	<p><i>Binary AND.</i> <code>\$leftValue -band \$rightValue</code></p> <p>Returns a number where bits are set to 1 if the bits of the lefthand and righthand arguments at that position are both 1. All other bits are set to 0. For example:</p> <pre>PS >\$boolean1 = "110110110" PS >\$boolean2 = "010010010" PS >\$int1 = [Convert]::ToInt32(\$boolean1, 2) PS >\$int2 = [Convert]::ToInt32(\$boolean2, 2) PS >\$result = \$int1 -band \$int2 PS >[Convert]::ToString(\$result, 2) 10010010</pre>
<code>-bor</code>	<p><i>Binary OR.</i> <code>\$leftValue -bor \$rightValue</code></p> <p>Returns a number where bits are set to 1 if either of the bits of the lefthand and righthand arguments at that position is 1. All other bits are set to 0. For example:</p> <pre>PS >\$boolean1 = "110110110" PS >\$boolean2 = "010010010" PS >\$int1 = [Convert]::ToInt32(\$boolean1, 2) PS >\$int2 = [Convert]::ToInt32(\$boolean2, 2) PS >\$result = \$int1 -bor \$int2 PS >[Convert]::ToString(\$result, 2) 110110110</pre>
<code>-bxor</code>	<p><i>Binary Exclusive OR.</i> <code>\$leftValue -bxor \$rightValue</code></p> <p>Returns a number where bits are set to 1 if either of the bits of the lefthand or righthand arguments at that position is 1, but not if both are. All other bits are set to 0. For example:</p> <pre>PS >\$boolean1 = "110110110" PS >\$boolean2 = "010010010" PS >\$int1 = [Convert]::ToInt32(\$boolean1, 2) PS >\$int2 = [Convert]::ToInt32(\$boolean2, 2) PS >\$result = \$int1 -bxor \$int2</pre>

Operator	Meaning
	<p>Returns a string, where the format items in the format string have been replaced with the text equivalent of the values in the value array.</p> <p>For example:</p> <pre>PS >"{0:n0}" -f 1000000000 1,000,000,000</pre> <p>The format string for the format operator is exactly the format string supported by the <code>.NET <i>String.Format</i></code> method.</p> <p>For more details about the syntax of the format string, see Chapter 9.</p>
<code>-as</code>	<p>The <i>type conversion operator</i>:</p> <pre><i>\$value -as [Type]</i></pre> <p>Returns <i>\$value</i> cast to the given .NET type. If this conversion is not possible, PowerShell returns <code>\$null</code>.</p> <p>For example:</p> <pre>PS >3/2 -as [int] 2 PS >\$result = "Hello" -as [int] PS >\$result -eq \$null True</pre>





2.11. Comparison Operators

The PowerShell comparison operators, listed in [Table 2-9](#), let you compare expressions against each other. By default, PowerShell's comparison operators are case insensitive. For all operators where case sensitivity applies, the `-i` prefix makes this case insensitivity explicit, while the `-c` prefix performs a case-sensitive comparison.

Table 2-9. Windows PowerShell comparison operators

Operator	Meaning
<code>-eq</code>	<p>The <i>equality operator</i>. <code>\$leftValue -eq \$rightValue</code></p> <p>For all primitive types, returns <code>\$true</code> if <code>\$leftValue</code> and <code>\$rightValue</code> are equal.</p> <p>When used with arrays, returns all elements in <code>\$leftValue</code> that are equal to <code>\$rightValue</code>.</p> <p>When used with any other type, PowerShell uses that type's <code>Equals()</code> method if it implements one.</p>
<code>-ne</code>	<p>The <i>negated equality operator</i>. <code>\$leftValue -ne \$rightValue</code></p> <p>For all primitive types, returns <code>\$true</code> if <code>\$leftValue</code> and <code>\$rightValue</code> are not equal.</p> <p>When used with arrays, returns all elements in <code>\$leftValue</code> that are not equal to <code>\$rightValue</code>.</p> <p>When used with any other type, PowerShell returns the negation of that type's <code>Equals()</code> method if it implements one.</p>
<code>-ge</code>	<p>The <i>greater-than-or-equal to operator</i>. <code>\$leftValue -ge \$rightValue</code></p> <p>For all primitive types, returns <code>\$true</code> if <code>\$leftValue</code> is greater than or equal to <code>\$rightValue</code>.</p> <p>When used with arrays, returns all elements in <code>\$leftValue</code> that are greater than or equal to <code>\$rightValue</code>.</p> <p>When used with any other type, PowerShell returns the result of that object's <code>Compare()</code> method if it implements one. If the method returns a number greater than or equal to 0, the operator returns <code>\$true</code>.</p>
<code>-gt</code>	<p>The <i>greater-than operator</i>. <code>\$leftValue -gt \$rightValue</code></p> <p>For all primitive types, returns <code>\$true</code> if <code>\$leftValue</code> is greater than <code>\$rightValue</code>.</p> <p>When used with arrays, returns all elements in <code>\$leftValue</code> that are greater than <code>\$rightValue</code>.</p>

Operator	Meaning
	<p>When used with any other type, PowerShell returns the result of that object's <code>Compare()</code> method if it implements one. If the method returns a number greater than 0, the operator returns <code>\$true</code>.</p>
<p><code>-lt</code></p>	<p>The <i>less-than operator</i>. <code>\$leftValue -lt \$rightValue</code></p> <p>For all primitive types, returns <code>\$true</code> if <code>\$leftValue</code> is less than <code>\$rightValue</code>.</p> <p>When used with arrays, returns all elements in <code>\$leftValue</code> that are less than <code>\$rightValue</code>.</p> <p>When used with any other type, PowerShell returns the result of that object's <code>Compare()</code> method if it implements one. If the method returns a number less than 0, the operator returns <code>\$true</code>.</p>
<p><code>-le</code></p>	<p>The <i>less-than-or-equal to operator</i>. <code>\$leftValue -le \$rightValue</code></p> <p>For all primitive types, returns <code>\$true</code> if <code>\$leftValue</code> is less than or equal to <code>\$rightValue</code>.</p> <p>When used with arrays, returns all elements in <code>\$leftValue</code> that are less than or equal to <code>\$rightValue</code>.</p> <p>When used with any other type, PowerShell returns the result of that object's <code>Compare()</code> method if it implements one. If the method returns a number less than or equal to 0, the operator returns <code>\$true</code>.</p>
<p><code>-like</code></p>	<p>The <i>like operator</i>. <code>\$leftValue -like Pattern</code></p> <p>Evaluates the pattern against the target, returning <code>\$true</code> if the simple match is successful.</p> <p>When used with arrays, returns all elements in <code>\$leftValue</code> that match <code>Pattern</code>.</p> <p>The <code>-like</code> operator supports these simple wildcard characters:</p> <ul style="list-style-type: none"> ? Any single unspecified character * Zero or more unspecified characters <p>[a-b]</p> <p>Any character in the range of a-b</p> <p>[ab]</p>

Operator	Meaning
	<p>The specified characters a or b</p> <p>For example: <pre>PS >"Test" -like "[A-Z]e?[tr]" True</pre> </p>
<code>-notlike</code>	The <i>negated like operator</i> : Returns <code>\$true</code> when the <code>-like</code> operator would return <code>\$false</code> .
<code>-match</code>	<p>The <i>match operator</i>: <pre>"target" -match Regular Expression</pre></p> <p>Evaluates the regular expression against the target, returning <code>\$true</code> if the match is successful. Once complete, PowerShell places the successful matches in the <code>\$matches</code> variable.</p> <p>When used with arrays, returns all elements in <i>Target</i> that match <i>Regular Expression</i>.</p> <p>The <code>\$matches</code> variable is a hashtable that maps the individual matches to the text they match. <code>0</code> is the entire text of the match, <code>1</code> and on contain the text from any unnamed captures in the regular expression, and string values contain the text from any named captures in the regular expression.</p> <p>For example: <pre>PS >"Hello World" -match "(.*) (.*)" True PS >\$matches[1] Hello</pre> </p> <p>For more details on regular expressions, see Chapter 3.</p>
<code>-notmatch</code>	<p>The <i>negated match operator</i>:</p> <p>Returns <code>\$true</code> when the <code>-match</code> operator would return <code>\$false</code>.</p> <p>The <code>-notmatch</code> operator still populates the <code>\$matches</code> variable with the results of <code>match</code>.</p>
<code>-contains</code>	<p>The <i>contains operator</i>: <pre>\$list -contains \$value</pre></p> <p>Returns <code>\$true</code> if the list specified by <code>\$list</code> contains the value <code>\$value</code>, that is, if <code>\$item -eq \$value</code> returns <code>\$true</code> for at least one item in the list.</p>
<code>-notcontains</code>	The <i>negated contains operator</i> : Returns <code>\$true</code> when the <code>-contains</code> operator would return <code>\$false</code> .
<code>-is</code>	<p>The <i>type operator</i>: <pre>\$leftValue -is type]</pre></p> <p>Returns <code>\$true</code> if <code>\$value</code> is (or extends) the specified .NET type.</p>
<code>-isnot</code>	The <i>negated type operator</i> : Returns <code>\$true</code> when the <code>-is</code> operator would return <code>\$false</code> .

2.12. Conditional Statements

Conditional statements in PowerShell let you change the flow of execution in your script.

2.12.1. if, elseif, and else Statements

```
if(condition)
{
    statement block
}
elseif(condition)
{
    statement block
}
else
{
    statement block
}
```

If *condition* evaluates to `$true`, then PowerShell executes the statement block you provide. Then, it resumes execution at the end of the `if / elseif / else` statement list. PowerShell requires the enclosing braces around the statement block even if the statement block contains only one statement.



See "Simple Operators" and "Comparison Operators," both earlier in this chapter, for a discussion of how PowerShell evaluates expressions as conditions.

If *condition* evaluates to `$false`, then PowerShell evaluates any following (optional) `elseif` conditions until one matches. If one matches, PowerShell executes the statement block associated with that condition, then resumes execution at the end of the `if / elseif / else` statement list.

For example:

```
$textToMatch = Read-Host "Enter some text"
$matchType = Read-Host "Apply Simple or Regex matching?"
$pattern = Read-Host "Match pattern"
if($matchType -eq "Simple")
{
    $textToMatch -like $pattern
}
elseif($matchType -eq "Regex")
{
    $textToMatch -match $pattern
}
else
{
    Write-Host "Match type must be Simple or Regex"
}
```

If none of the conditions evaluate to `$true`, then PowerShell executes the statement block associated with the (optional) `else` clause, then resumes execution at the end of the `if / elseif / else` statement list.

2.12.2. switch Statements

```
switch options expression
{
    comparison value          { statement block }
    -or-
    { comparison expression } { statement block }

    (...)
    default                   { statement block }
}
```

or:

```
switch options -file filename
{
    comparison value          { statement block }
    -or-
    { comparison expression } { statement block }

    (...)
    default                   { statement block }
}
```

When PowerShell evaluates a `switch` statement, it evaluates `expression` against the statements in the switch body. If `expression` is a list of values, PowerShell evaluates each item against the statements in the switch body. If you specify the `-file` option, PowerShell treats the lines in the file as though they were a list of items in `expression`.

The `comparison value` statements let you match the current input item against the pattern specified by `comparison value`. By default, PowerShell treats this as a case-insensitive exact match, but the options you provide to the `switch` statement can change this, as shown in [Table 2-10](#).

Table 2-10. Options supported by PowerShell switch statements

Option	Meaning
<code>-casesensitive</code>	<i>Case-sensitive match.</i>
<code>-c</code>	With this option active, PowerShell executes the associated statement block only if the current input item exactly matches the value specified by <code>comparison value</code> . If the current input object is a string, the match is case-sensitive.
<code>-exact</code>	<i>Exact match.</i>
<code>-e</code>	With this option active, PowerShell executes the associated statement block only if the current input item exactly matches the value specified by <code>comparisonvalue</code> . This match is case-insensitive. This is the default mode of operation.
<code>-regex</code>	<i>Regular-expression match.</i>
<code>-r</code>	With this option active, PowerShell executes the associated statement block only if the current input item matches the regular expression specified by <code>comparisonvalue</code> . This match is case-insensitive.

Option	Meaning
<code>-wildcard</code>	<i>Wildcard match.</i>
<code>-w</code>	<p>With this option active, PowerShell executes the associated statement block only if the current input item matches the wildcard specified by <i>comparisonvalue</i>.</p> <p>The wildcard match supports the following simple wildcard characters:</p> <ul style="list-style-type: none"> ? Any single unspecified character * Zero or more unspecified characters <p>[a-b]</p> <p>Any character in the range of a-b</p> <p>[ab]</p> <p>The specified characters a or b</p> <p>This match is case-insensitive.</p>

The { *comparison expression* } statements let you process the current input item (stored in the `$_` variable) in an arbitrary script block. When PowerShell processes a { *comparison expression* } statement, it executes the associated statement block only if { *comparison expression* } evaluates to `$true`.

PowerShell executes the statement block associated with the (optional) `default` statement if no other statements in the `switch` body match.

When processing a `switch` statement, PowerShell tries to match the current input object against each statement in the `switch` body, falling through to the next statement even after one or more have already matched. To have PowerShell exit a `switch` statement after it processes a match, include a `break` statement as the last statement in the statement block.

For example:

```
$myPhones = "(555) 555-1212","555-1234"

switch -regex ($myPhones)
{
  { $_.Length -le 8 } { "Area code was not specified";
    break }
  { $_.Length -gt 8 } { "Area code was specified" }
  "\\((555)\\).*"      { "In the $($matches[1]) area code" }
}
```

Produces the output:

```
Area code was specified  
In the 555 area code  
Area code was not specified
```



See the following section, "Looping Statements," for more information about the `break` statement.

By default, PowerShell treats this as a case-insensitive exact match, but the options you provide to the `switch` statement can change this.



2.13. Looping Statements

Looping statements in PowerShell let you execute groups of statements multiple times.

2.13.1. for Statement

```
:loop_label for(initialization; condition; increment)
{
    statement block
}
```

When PowerShell executes a *for* statement, it first executes the expression given by *initialization*. It next evaluates *condition*. If *condition* evaluates to *\$true*, PowerShell executes the given statement block. It then executes the expression given by *increment*. PowerShell continues to execute the statement block and *increment* statement as long as *condition* evaluates to *\$true*.

For example:

```
for($counter = 0; $counter -lt 10; $counter++)
{
    Write-Host "Processing item $counter"
}
```

The *break* and *continue* statements (discussed later in the chapter) can specify the *loop_label* of any enclosing looping statement as their target.

2.13.2. foreach Statement

```
:loop_label foreach(variable in expression)
{
    statement block
}
```

When PowerShell executes a *foreach* statement, it executes the pipeline given by *expression*—for example, `Get-Process|Where-Object{$_ .Handles -gt 500 }` or `1..10`. For each item produced by the expression, it assigns that item to the variable specified by *variable* and then executes the given statement block. For example:

```
$handleSum = 0;
foreach($process in Get-Process |
    Where-Object { $_.Handles -gt 500 })
{
    $handleSum += $process.Handles
}
$handleSum
```

The *break* and *continue* statements (discussed later in the chapter) can specify the *loop_label* of any enclosing looping statement as their target.

2.13.3. while Statement

```
:loop_label while(condition)
```

```
{
    statement block
}
```

When PowerShell executes a `while` statement, it first evaluates the expression given by *condition*. If this expression evaluates to `$true`, PowerShell executes the given statement block. PowerShell continues to execute the statement block as long as *condition* evaluates to `$true`. For example:

```
$command = "";
while($command -notmatch "quit")
{
    $command = Read-Host "Enter your command"
}
```

The `break` and `continue` statements (discussed later in this chapter) can specify the *loop_label* of any enclosing looping statement as their target.

2.13.4. do ... while Statement/do ... until Statement

```
:loop_label do
{
    statement block
} while(condition)
```

or:

```
:loop_label do
{
    statement block
} until(condition)
```

When PowerShell executes a `do...while` or `do...until` statement, it first executes the given statement block. In a `do...while` statement, PowerShell continues to execute the statement block as long as *condition* evaluates to `$true`. In a `do...until` statement, PowerShell continues to execute the statement as long as *condition* evaluates to `$false`. For example:

```
$validResponses = "Yes","No"
$response = ""
do
{
    $response = Read-Host "Yes or No?"
} while($validResponses -notcontains $response)
"Got it."
$response = ""
do
{
    $response = Read-Host "Yes or No?"
} until($validResponses -contains $response)
"Got it."
```

The `break` and `continue` statements (discussed later in this chapter) can specify the *loop_label* of any enclosing looping statement as their target.

2.13.5. Flow Control Statements

PowerShell supports two statements to help you control flow within loops: `break` and `continue`.

2.13.5.1. break

The `break` statement halts execution of the current loop. PowerShell then resumes execution at the end of the current looping statement, as though the looping statement had completed naturally. If you specify a label with the `break` statement—for example, `break :outer_loop`—PowerShell halts the execution of that loop instead.

For example:

```
:outer for($counter = 0; $counter -lt 5; $counter++)
{
    for($counter2 = 0; $counter2 -lt 5; $counter2++)
    {
        if($counter2 -eq 2)
        {
            break :outer
        }

        Write-Host "Processing item $counter,$counter2"
    }
}
```

Produces the output:

```
Processing item 0,0
Processing item 0,1
Processing item 1,0
Processing item 1,1
Processing item 2,0
Processing item 2,1
Processing item 3,0
Processing item 3,1
Processing item 4,0
Processing item 4,1
```

2.13.5.2. continue

The `continue` statement skips execution of the rest of the current statement block. PowerShell then continues with the next iteration of the current looping statement as though the statement block had completed naturally. If you specify a label with the `continue` statement—for example, `continue :outer`—PowerShell continues with the next iteration of that loop instead.

For example:

```
:outer for($counter = 0; $counter -lt 5; $counter++)
{
    for($counter2 = 0; $counter2 -lt 5; $counter2++)
    {
        if($counter2 -eq 2)
        {
            continue :outer
        }
    }
}
```

```
        Write-Host "Processing item $counter,$counter2"  
    }  
}
```

Produces the output:

```
Processing item 0,0  
Processing item 0,1  
Processing item 0,3  
Processing item 0,4  
Processing item 1,0  
Processing item 1,1  
Processing item 1,3  
Processing item 1,4  
Processing item 2,0  
Processing item 2,1  
Processing item 2,3  
Processing item 2,4  
Processing item 3,0  
Processing item 3,1  
Processing item 3,3  
Processing item 3,4  
Processing item 4,0  
Processing item 4,1  
Processing item 4,3  
Processing item 4,4
```





2.14. Working with the .NET Framework

One feature that gives PowerShell its incredible reach into both system administration and application development is its capability to leverage Microsoft's enormous and broad .NET Framework.

Work with the .NET Framework in PowerShell comes mainly by way of one of two tasks: calling methods or accessing properties.

2.14.1. Static Methods

To call a static method on a class, type:

```
[ClassName]::MethodName(parameter list)
```

For example:

```
PS >[System.Diagnostics.Process]::GetProcessById(0)
```

gets the process with the ID of 0 and displays the following output:

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
0	0	0	16	0		0	Idle

2.14.2. Instance Methods

To call a method on an instance of an object, type:

```
$objectReference.MethodName(parameter list)
```

For example:

```
PS >$process = [System.Diagnostics.Process]::  
GetProcessById(0)  
PS >$process.Refresh()
```

This stores the process with the ID of 0 into the `$process` variable. It then calls the `Refresh()` instance method on that specific process.

2.14.3. Static Properties

To access a static property on a class, type:

```
[ClassName]::PropertyName
```

or:

```
[ClassName]::PropertyName = value
```

For example, the `[System.DateTime]` class provides a `Now` static property that returns the current time:

```
PS >[System.DateTime]::Now
Sunday, July 16, 2006 2:07:20 PM
```

Although rare, some types let you set the value of some static properties.

2.14.4. Instance Properties

To access an instance property on an object, type:

```
$objectReference.PropertyName
```

or:

```
$objectReference.PropertyName = value
```

For example:

```
PS >$today = [System.DateTime]::Now
PS >$today.DayOfWeek
Sunday
```

This stores the current date in the `$today` variable. It then calls the `DayOfWeek` instance property on that specific date.

2.14.5. Learning About Types

The two primary avenues for learning about classes and types are the `Get-Member` cmdlet and the documentation for the .NET Framework.

2.14.5.1. The Get-Member cmdlet

To learn what methods and properties a given type supports, pass it through the `Get-Member` cmdlet, as shown in Table 2-11.

Table 2-11. Working with the Get-Member cmdlet


Action	Result
<code>[typename] Get-Member -Static</code>	All the static methods and properties of a given type.
<code>\$objectReference Get-Member -Static</code>	All the static methods and properties provided by the type in <code>\$objectReference</code> .
<code>\$objectReference Get-Member</code>	All the instance methods and properties provided by the type in <code>\$objectReference</code> . If <code>\$objectReference</code> represents a collection of items, PowerShell returns the instances and properties of the types contained by that collection. To view the

Action	Result
	instances and properties of a collection itself, use the <code>-InputObject</code> parameter of <code>Get-Member</code> : <code>Get-Member -InputObject \$objectReference</code>
<code>[typename] Get-Member</code>	All the instance methods and properties of a <code>System.RuntimeType</code> object that represents this type.

2.14.5.2. .NET Framework documentation

Another source of information about the classes in the .NET Framework is the documentation itself, available through the search facilities at <http://msdn.microsoft.com>.

Typical documentation for a class first starts with a general overview, then provides a hyperlink to the members of the class—listing the methods and properties it supports.



To get to the documentation for the members quickly, search for them more explicitly by adding the term "members" to your MSDN search term:

`classname members`

The documentation for the members of a class lists their constructors, methods, properties, and more. It uses an S icon to represent the static methods and properties. Click the member name for more information about that member—including the type of object that the member produces.

2.14.6. Type Shortcuts

When you specify a type name, PowerShell lets you use a short form for some of the most common types, as listed in Table 2-12.

Table 2-12. PowerShell type shortcuts

Type shortcut	Full classname
<code>[Adsi]</code>	<code>[System.DirectoryServices.DirectoryEntry]</code>
<code>[Hashtable]</code>	<code>[System.Collections.Hashtable]</code>
<code>[PSObject]</code>	<code>[System.Management.Automation.PSObject]</code>
<code>[Ref]</code>	<code>[System.Management.Automation.PSReference]</code>
<code>[Regex]</code>	<code>[System.Text.RegularExpressions.Regex]</code>
<code>[ScriptBlock]</code>	<code>[System.Management.Automation.ScriptBlock]</code>
<code>[Switch]</code>	<code>[System.Management.Automation.SwitchParameter]</code>

Type shortcut	Full classname
[Wmi]	[System.Management.ManagementObject]
[WmiClass]	[System.Management.ManagementClass]
[WmiSearcher]	[System.Management.ManagementObjectSearcher]
[Xml]	[System.Xml.XmlDocument]
[TypeName]	[System.TypeName]

2.14.7. Creating Instances of Types

```
$ObjectReference = New-Object TypeName parameters
```

Although static methods and properties of a class generate objects, you will often want to create them explicitly yourself. PowerShell's `New-Object` cmdlet lets you create an instance of the type you specify. The parameter list must match the list of parameters accepted by one of the type's constructors, as documented on MSDN.

For example:

```
$webClient = New-Object Net.WebClient
$webClient.DownloadString("http://search.msn.com")
```

Most common types are available by default. However, many are available only after you load the library (called the *assembly*) that defines them. The MSDN documentation for a class includes the assembly that defines it.

To load an assembly, use the methods provided by the `System.Reflection.Assembly` class:

```
PS > [Reflection.Assembly]::LoadWithPartialName("System.Web")
```

```
GAC    Version      Location
--    -
True   v2.0.50727     C:\WINDOWS\assembly\GAC_32\...\
                System.Web.dll
```

```
PS > [Web.HttpUtility]::UrlEncode("http://search.msn.com")
http%3a%2f%2fsearch.msn.com
```



The `LoadWithPartialName` method is unsuitable for scripts that you want to share with others or use in a production environment. It loads the most current version of the assembly, which may not be the same as the version you used to develop your script. To load an assembly in the safest way possible, use its fully qualified name with the `[Reflection.Assembly]::Load()` method.

2.14.8. Interacting with COM Objects

PowerShell lets you access methods and properties on COM objects the same way you would interact with objects from the .NET Framework. To interact with a COM object, use its `ProgId` with the `-ComObject` parameter (often shortened to `-Com`) on `New-Object`:

```
PS >$shell = New-Object -Com Shell.Application
PS >$shell.Windows() | Select-Object
LocationName,LocationUrl
```

For more information about the COM objects most useful to system administrators, see [Chapter 6](#).

2.14.9. Extending Types

PowerShell supports two ways to add your own methods and properties to any type: the `Add-Member` cmdlet and a custom types extension file.

2.14.9.1. The Add-Member cmdlet

The `Add-Member` cmdlet lets you dynamically add methods, properties, and more to an object. It supports the extensions shown in [Table 2-13](#).

Table 2-13. Selected member types supported by the Add-Member cmdlet

Member type	Meaning
AliasProperty	A property defined to alias another property: <pre>PS >\$testObject = [PsObject] "Test" PS >\$testObject Add-Member "AliasProperty" Count Length PS >\$testObject.Count 4</pre>
CodeProperty	A property defined by a <code>System.Reflection.MethodInfo</code> . This method must be public, static, return results (nonvoid), and take one parameter of type <code>PsObject</code> .
NoteProperty	A property defined by the initial value you provide: <pre>PS >\$testObject = [PsObject] "Test" PS >\$testObject Add-Member NoteProperty Reversed tseT PS >\$testObject.Reversed tseT</pre>
ScriptProperty	A property defined by the script block you provide. In that script block, <code>\$this</code> refers to the current instance: <pre>PS >\$testObject = [PsObject] ("Hi" * 100) PS >\$testObject Add-Member ScriptProperty IsLong { >> \$this.Length -gt 100 >> } >> \$testObject.IsLong >> True</pre>
PropertySet	A property defined as a shortcut to a set of properties. Used in cmdlets such as <code>Select-Object</code> : <pre>PS >\$testObject = [PsObject] [DateTime]::</pre>

Member type	Meaning
	<pre> Now PS >\$collection = New-Object ' >> Collections.ObjectModel. Collection'[System.String] >> \$collection.Add("Month") >> \$collection.Add("Year") >> \$testObject Add-Member PropertySet MonthYear \$collection >> \$testObject select MonthYear >> Month Year ----- 6 2007 </pre>
CodeMethod	A method defined by a <code>System.Reflection.MethodInfo</code> . This method must be public, static, and take one parameter of type <code>PsObject</code> .
ScriptMethod	<p>A method defined by the script block you provide. In that script block, <code>\$this</code> refers to the current instance, and <code>\$args</code> refers to the input parameters:</p> <pre> PS >\$testObject = [PsObject] "Hello" PS >\$testObject Add-Member ScriptMethod IsLong { >> \$this.Length -gt \$args[0] >> } >> \$testObject.IsLong(3) >> \$testObject.IsLong(100) >> True False </pre>

2.14.9.2. Custom type extension files

While the `Add-Member` cmdlet lets you customize individual objects, PowerShell also supports configuration files that let you customize all objects of a given type. For example, you might want to add a `Reverse()` method to all strings or a `HelpUrl` property (based on the MSDN *Url Aliases*) to all types.

PowerShell adds several type extensions to the file `types.ps1xml`, in the PowerShell installation directory. This file is useful as a source of examples, but you should not modify it directly. Instead, create a new one and use the `Update-TypeData` cmdlet to load your customizations. The following command loads `Types.custom.ps1xml` from the same directory as your profile:

```

$typesFile = Join-Path (Split-Path $profile) "Types.
Custom.Ps1Xml"
Update-TypeData -PrependPath $typesFile

```





2.15. Writing Scripts, Reusing Functionality

When you want to start packaging and reusing your commands, the best place to put them is in scripts and functions. A *script* is a text file that contains a sequence of PowerShell commands. A *function* is also a sequence of PowerShell commands but is usually used within a script to break it into smaller, more easily understood segments.

2.15.1. Writing Scripts

To write a script, write your PowerShell commands in a text editor and save the file with a `.ps1` extension.

2.15.2. Running Scripts

There are two ways to execute a script: by invoking it or by dot-sourcing it.

2.15.2.1. Invoking

Invoking a script runs the commands inside it. Unless explicitly defined with the `GLOBAL` scope keyword, variables and functions defined in the script do not persist once the script exits.

You invoke a script by using the invoke/call operator (`&`) with the script name as the parameter:

```
& "C:\Script Directory\Run-Commands.ps1" Parameters
```

You can use either a fully qualified path or a path relative to the current location. If the script is in the current directory, you must explicitly say so:

```
.\Run-Commands.ps1 Parameters
```

If the path contains no spaces, you may omit both the quotes and invoke the operator.

2.15.2.2. Dot-sourcing

Dot-sourcing a script runs the commands inside it. Unlike invoking a script, variables and functions defined in the script *do* persist after the script exits.


You dot-source a script by using the dot operator (`.`) and providing the script name as the parameter:

```
. "C:\Script Directory\Run-Commands.ps1" Parameters
```

You can use either a fully qualified path or a path relative to the current location. If the script is in the current directory, you must explicitly say so:

```
. .\Run-Commands.ps1 Parameters
```

If the path contains no spaces, you may omit the quotes.



By default, a security feature in PowerShell called the Execution Policy prevents scripts from running. When you want to enable scripting in PowerShell, you must change this setting. To understand the different execution policies available to you, type **Get-Help about_signing**. After selecting an execution policy, use the **Set-ExecutionPolicy** cmdlet to configure it:

```
Set-ExecutionPolicy RemoteSigned
```

2.15.3. Providing Input to Scripts

PowerShell offers several options for processing input to a script.

2.15.3.1. Argument array

To access the command-line arguments by position, use the argument array that PowerShell places in the `$args` special variable:

```
$firstArgument = $args[0]
$secondArgument = $args[1]
$argumentCount = $args.Count
```


2.15.3.2. Formal parameters

```
param([TypeName] $variableName = Default, ...)
```

Formal parameters let you benefit from some of the many benefits of PowerShell's consistent command-line parsing engine.

PowerShell exposes your parameter names (for example, `$variableName`) the same way that it exposes parameters in cmdlets. Users need only to type enough of your parameter name to disambiguate it from the rest of the parameters. If the user does not specify the parameter name, PowerShell attempts to assign the input to your parameters by position.

If you specify a type name for the parameter, PowerShell ensures that the user input is of that type. If you specify a default value, PowerShell uses that value if the user does not provide input for that parameter.



To make a parameter mandatory, define the default value so that it throws an error:

```
param($mandatory =
    $(throw "This parameter is required."))
```

2.15.3.3. Pipeline input

To access the data being passed to your script via the pipeline, use the input enumerator that PowerShell places in the `$input` special variable:

```
foreach($element in $input)
{
    "Input was: $element"
}
```


The `$input` variable is a .NET enumerator over the pipeline input. Enumerators support streaming scenarios very efficiently but do not let you access arbitrary elements as you would with an array. If you want to process their elements again, you must call the `Reset()` method on the `$input` enumerator once you reach the end.

If you need to access the pipeline input in an unstructured way, use the following command to convert the input enumerator to an array:

```
$inputArray = @($input)
```

2.15.3.4. Cmdlet keywords in scripts

When pipeline input is a core scenario of your script, you may include statement blocks labeled `begin`, `process`, and `end`:

```
param(...)
```

```
begin
{
    ...
}
process
{
    ...
}
end
{
    ...
}
```

PowerShell executes the `begin` statement when it loads your script, the `process` statement for each item passed down the pipeline, and the `end` statement after all pipeline input has been processed. In the `process` statement block, the `$_` variable represents the current pipeline object.

When you write a script that includes these keywords, all the commands in your script must be contained within the statement blocks.

2.15.3.5. `$MyInvocation` automatic variable

The `$MyInvocation` automatic variable contains information about the context under which the script was run, including detailed information about the command (`MyCommand`), the script that defines it (`ScriptName`), and more.

2.15.4. Retrieving Output from Scripts

PowerShell provides three primary ways to retrieve output from a script.

2.15.4.1. Pipeline output

any command

The return value/output of a script is any data that it generates but does not capture. If a script contains the commands:

```
"Text Output"  
5*5
```

then assigning the output of that script to a variable creates an array with the two values, `Text Output` and `25`.

2.15.4.2. Return statement

```
return value
```

The statement

```
return $false
```

is simply a short form for pipeline output:


```
$false  
return
```

2.15.4.3. Exit statement

```
exit errorLevel
```

The `exit` statement returns an error code from the current script or instance of PowerShell. If called anywhere in a script (inline, in a function, or in a script block), it exits the script.

If called outside of a script, it exits PowerShell. The `exit` statement sets the `$LastExitCode` automatic variable to `errorLevel`. In turn, that sets the `$?` automatic variable to `$false` if `errorLevel` is not zero.



See [Chapter 4](#) for more information about automatic variables.

2.15.5. Functions

```
function SCOPE:name(parameters)  
{  
    statement block  
}
```

or:

```
filter SCOPE:name(parameters)  
{  
    statement block  
}
```

Functions let you package blocks of closely related commands into a single unit that you can access by name.

Valid scope names are `global` (to create a function available to the entire shell), `script` (to create a function available only to the current script), `local` (to create a function available only to the current scope and subscopes), and `private` (to create a function available only to the current scope). The default scope is the `local` scope, which follows the same rules as those of default variable scopes.

The content of a function's statement block follows the same rules as the content of a script. Functions support the `$args` array, formal parameters, the `$input` enumerator, cmdlet keywords, pipeline output, and equivalent return semantics.

A common mistake is to call a function as you would call a method:

```
$result = GetMyResults($item1, $item2)
```

PowerShell treats functions as it treats scripts and other commands, so this should instead be:

```
$result = GetMyResults $item1 $item2
```

The first command passes an array that contains the items `$item1` and `$item2` to the `GetMyResults` function.

A parameter declaration, as an alternative to a `param` statement, follows the same syntax as the formal parameter list but does not require the `param` keyword.

A filter is simply a function where the statements are treated as though they are contained within a `process` statement block.

Commands in your script can access only functions that have already been defined. This can often make large scripts difficult to understand when the beginning of the script is composed entirely of helper functions. Structuring a script in the following manner often makes it more clear:

```
function Main
{
    (...)
    HelperFunction
    (...)
}

function HelperFunction
{
    (...)
}

. Main
```

As with a script, you may either invoke or dot-source a function.

2.15.6. Script Blocks

```
$objectReference =
{
    statement block
```

```
}
```

PowerShell supports script blocks, which act exactly like unnamed functions and scripts. Like both scripts and functions, the content of a script block's statement block follows the same rules as the content of a function or script. Script blocks support the `$args` array, formal parameters, the `$input` enumerator, cmdlet keywords, pipeline output, and equivalent return semantics.

As with both scripts and functions, you may either invoke or dot-source a script block. Since a script block does not have a name, you either invoke it directly (`& { "Hello" }`) or invoke the variable (`& $objectReference`) that contains it.



2.16. Managing Errors

PowerShell supports two classes of errors: nonterminating and terminating. It collects both types of errors as a list in the `$error` automatic variable.

2.16.1. Nonterminating Errors

Most errors are *nonterminating errors*, in that they do not halt execution of the current cmdlet, script, function, or pipeline. When a command outputs an error (via PowerShell's error-output facilities), PowerShell writes that error to a stream called the *error output stream*.

You can output a nonterminating error using the `Write-Error` cmdlet (or the `WriteError()` API when writing a cmdlet).

The `$ErrorActionPreference` automatic variable lets you control how PowerShell handles nonterminating errors. It supports the following values, as shown in Table 2-14.

Table 2-14. `$ErrorActionPreference` automatic variable values

Value	Meaning
<code>SilentlyContinue</code>	Do not display errors.
<code>Stop</code>	Treat nonterminating errors as terminating errors.
<code>Continue</code>	Display errors, but continue execution of the current cmdlet, script, function, or pipeline. This is the default.
<code>Inquire</code>	Display a prompt that asks how PowerShell should treat this error.

Most cmdlets let you configure this explicitly by passing one of the above values to its `ErrorAction` parameter.

2.16.2. Terminating Errors

A *terminating error* halts execution of the current cmdlet, script, function, or pipeline. If a command (such as a cmdlet or .NET method call) generates a structured exception (for example, if you provide a method with parameters outside their valid range), PowerShell exposes this as a terminating error. PowerShell also generates a terminating error if it fails to parse an element of your script, function, or pipeline.

You can generate a terminating error in your script using the `throw` keyword:

```
throw message
```



In your own scripts and cmdlets, generate terminating errors only when the fundamental intent of the operation is impossible to accomplish. For example, failing to execute a command on a remote server should be considered a nonterminating error, while failing to connect to the remote server altogether should be considered a terminating error.

PowerShell lets you intercept terminating errors if you define a `trap` statement before PowerShell encounters that error:

```
trap [exception type]
{
    statement block
    [continue or break]
}
```

If you specify an exception type, the `trap` statement applies only to terminating errors of that type.

If specified, the `continue` keyword tells PowerShell to continue processing the rest of your script, function, or pipeline after the point at which it encountered the terminating error.

If specified, the `break` keyword tells PowerShell to halt processing the rest of your script, function, or pipeline after the point at which it encountered the terminating error. `Break` is the default mode and applies if you specify neither `break` nor `continue` at all.



2.17. Formatting Output

Pipeline / Formatting Command

When objects reach the end of the output pipeline, PowerShell converts them to text to make them suitable for human consumption. PowerShell supports several options to help you control this formatting process, as listed in Table 2-15.

Table 2-15. PowerShell formatting commands

Formatting command	Result
<code>Format-Table Properties</code>	<p>Formats the properties of the input objects as a table, including only the object properties you specify. If you do not specify a property list, PowerShell picks a default set.</p> <p>In addition to supplying object properties, you may also provide advanced formatting statements:</p> <pre>PS > Get-Process ` Format-Table -Auto Name, ` @{Label="HexId"; Expression={ "{0:x}" -f \$_.Id} Width=4 Align="Right" }</pre> <p>The advanced formatting statement is a hashtable with the keys <code>Label</code> and <code>Expression</code> (or any short form of them). The value of the <code>Expression</code> key should be a script block that returns a result for the current object (represented by the <code>\$_</code> variable).</p> <p>For more information about the <code>Format-Table</code> cmdlet, type <code>Get-Help Format-Table</code>.</p>
<code>Format-List Properties</code>	<p>Formats the properties of the input objects as a list, including only the object properties you specify. If you do not specify a property list, PowerShell picks a default set.</p> <p>The <code>Format-List</code> cmdlet supports the advanced formatting statements as used by the <code>Format-Table</code> cmdlet.</p> <p>The <code>Format-List</code> cmdlet is the one you will use most often to get a detailed summary of an object's properties.</p> <p>The command <code>Format-List *</code> returns all properties but does not include those that PowerShell hides by default. The command <code>Format-List * -Force</code> returns all properties.</p> <p>For more information about the <code>Format-List</code> cmdlet, type <code>Get-Help Format-List</code>.</p>
<code>Format-Wide Property</code>	<p>Formats the properties of the input objects in an extremely terse summary view. If you do not specify a property, PowerShell picks a default.</p> <p>In addition to supplying object properties, you may also provide advanced formatting</p>

Formatting command	Result
	<p>statements:</p> <pre data-bbox="315 281 743 401">PS >Get-Process ` Format-Wide -Auto ` @{ Expression={ "{0:x}" -f \$.Id} }</pre> <p>The advanced formatting statement is a hashtable with the key <code>Expression</code> (or any short form of it). The value of the <code>Expression</code> key should be a script block that returns a result for the current object (represented by the <code>\$_</code> variable).</p> <p>For more information about the <code>Format-Wide</code> cmdlet, type <code>Get-Help Format-Wide</code>.</p>

2.17.1. Custom formatting files

2.17.1.1. Custom formatting files

All the formatting defaults in PowerShell (e.g., when you do not specify a formatting command or formatting properties) are driven by the **.Format.Ps1Xml* files in the installation directory in a manner similar to the type extension files mentioned in the "Custom type extension files" section in "Working with the .NET Framework," earlier in this chapter.

To create your own formatting customizations, use these files as a source of examples, but do not modify them directly. Instead, create a new file and use the `Update-FormatData` cmdlet to load your customizations. The `Update-FormatData` cmdlet applies your changes to the current instance of PowerShell. If you wish to load them every time you launch PowerShell, call `Update-FormatData` in your profile script. The following command loads *Format.custom.Ps1Xml* from the same directory as your profile:

```
$formatFile = Join-Path (Split-Path $profile)
"Format.Custom.Ps1Xml"
Update-FormatData -PrependPath $typesFile
```





2.18. Capturing Output

There are several ways to capture the output of commands in PowerShell, as listed in [Table 2-16](#).

Table 2-16. Capturing output in PowerShell

Command	Result
<code>\$variable = Command</code>	Stores the objects produced by the PowerShell command into <code>\$variable</code> .
<code>\$variable = Command Out-String</code>	Stores the visual representation of the PowerShell command into <code>\$variable</code> . This is the PowerShell command after it's been converted to human-readable output.
<code>\$variable = NativeCommand</code>	Stores the (string) output of the native command into <code>\$variable</code> . PowerShell stores this as a list of strings—one for each line of output from the native command.
<code>Command -OutVariable variable</code>	For most commands, stores the objects produced by the PowerShell command into <code>\$variable</code> . The parameter <code>-OutVariable</code> can also be written <code>-Ov</code> .
<code>Command > File</code>	Redirects the visual representation of the PowerShell (or standard output of a native command) into <code>File</code> , overwriting <code>File</code> if it exists. Errors are not captured by this redirection.
<code>Command >> File</code>	Redirects the visual representation of the PowerShell (or standard output of a native command) into <code>File</code> , appending to <code>File</code> if it exists. Errors are not captured by this redirection.
<code>Command 2> File</code>	Redirects the errors from the PowerShell or native command into <code>File</code> , overwriting <code>File</code> if it exists.
<code>Command 2>> File</code>	Redirects the errors from the PowerShell or native command into <code>File</code> , appending to <code>File</code> if it exists.
<code>Command > File 2>&1</code>	Redirects both the error and standard output streams of the PowerShell or native command into <code>File</code> , overwriting <code>File</code> if it exists.
<code>Command >> File 2>&1</code>	Redirects both the error and standard output streams of the PowerShell or native command into <code>File</code> , appending to <code>File</code> if it exists.



2.19. Tracing and Debugging

The three facilities for tracing and debugging in PowerShell are the `Set-PsDebug` cmdlet, the `Trace-Command` cmdlet, and the verbose cmdlet output.

2.19.1. The Set-PsDebug Cmdlet

The `Set-PsDebug` cmdlet lets you control tracing, stepping, and strict mode in PowerShell. [Table 2-17](#) lists the parameters of the `Set-PsDebug` cmdlet.

Table 2-17. Parameters of the Set-PsDebug cmdlet

Parameter	Description
<code>Trace</code>	Sets the amount of tracing detail that PowerShell outputs when running commands. A value of <code>1</code> outputs all lines as PowerShell evaluates them. A value of <code>2</code> outputs all lines as PowerShell evaluates them, along with information about variable assignments, function calls, and scripts. A value of <code>0</code> disables tracing.
<code>Step</code>	Enables and disables per-command stepping. When enabled, PowerShell prompts you before it executes a command.
<code>Strict</code>	Enables and disables strict mode. When enabled, PowerShell throws a terminating error if you attempt to reference a variable that you have not yet defined.
<code>off</code>	Turns off tracing, stepping, and strict mode.

2.19.2. The Trace-Command Cmdlet

```
Trace-Command CommandDiscovery -PsHost { gci c:\ }
```

The `Trace-Command` cmdlet exposes diagnostic and support information for PowerShell commands. PowerShell groups its diagnostic information into categories called *trace sources*.

A full list of trace sources is available through the `Get-TraceSource` cmdlet.

For more information about the `Trace-Command` cmdlet, type `Get-Help Trace-Command`.

2.19.3. The Verbose Cmdlet Output

```
Cmdlet -Verbose
```

PowerShell commands can generate verbose output using the `Write-Verbose` cmdlet (if written as a script), or the `WriteVerbose()` API (when written as a cmdlet).

The `$VerbosePreference` automatic variable lets you control how PowerShell handles verbose output. It supports the values listed in [Table 2-18](#).

Table 2-18. VerbosePreference automatic variable values

Value	Meaning
<code>SilentlyContinue</code>	Do not display verbose output. This is the default.
<code>Stop</code>	Treat verbose output as a terminating error.
<code>Continue</code>	Display verbose output and continue execution of the current cmdlet, script, function, or pipeline.
<code>Inquire</code>	Display a prompt that asks how PowerShell should treat this verbose output.

Most cmdlets let you configure this explicitly by passing one of the values listed in [Table 2-18](#) to its `Verbose` parameter.



2.20. Common Customization Points

As useful as it is out of the box, PowerShell offers several avenues for customization and personalization.

2.20.1. Console Settings

The Windows PowerShell user interface offers several features to make your shell experience more efficient.

2.20.1.1. Adjust your window size

In the System menu (right-click the PowerShell icon at the top left of the console window), select Properties → Layout. The Window Size options let you control the actual window size (how big the window appears on screen), while the Screen Buffer Size options let you control the virtual window size (how much content the window can hold). If the screen buffer size is larger than the actual window size, the console window changes to include scrollbars. Increase the virtual window height to make PowerShell store more output from earlier in your session. If you launch PowerShell from the Start menu, PowerShell launches with some default modifications to the window size.

2.20.1.2. Make text selection easier

In the System menu, click Options → QuickEdit Mode. QuickEdit mode lets you use the mouse to efficiently copy and paste text into or out of your PowerShell console. If you launch PowerShell from the Start menu, PowerShell launches with QuickEdit mode enabled.

2.20.1.3. Use hotkeys to operate the shell more efficiently

The Windows PowerShell console supports many hotkeys that help make operating the console more efficient, as shown in [Table 2-19](#).

Table 2-19. Windows PowerShell hotkeys

Hotkey	Meaning
Windows key + r, and then type powershell	Launch Windows PowerShell.
Up arrow	Scan backward through your command history.
Down arrow	Scan forward through your command history.
Page Up	Display the first command in your command history.
Page Down	Display the last command in your command history.
Left arrow	Move cursor one character to the left on your command line.
Right arrow	Move cursor one character to the right on your command line. If at the end of the line, it inserts a character from the text of your last command at that position.
Home	Move the cursor to the beginning of the command line.
End	Move the cursor to the end of the command line.

Hotkey	Meaning
Control + left arrow	Move the cursor one word to the left on your command line.
Control + right arrow	Move the cursor one word to the right on your command line.
Alt + space, e, l	Scroll through the screen buffer.
Alt + space, e, f	Search for text in the screen buffer.
Alt + space, e, k	Select text to be copied from the screen buffer.
Alt + space, e, p	Paste clipboard contents into the Windows PowerShell console.
Alt + space, c	Close the Windows PowerShell console.
Control + c	Cancel the current operation.
Control + break	Forcefully close the Windows PowerShell window.
Control + home	Delete characters from the beginning of the current command line up to (but not including) the current cursor position.
Control + end	Delete characters from (and including) the current cursor position to the end of the current command line.
F1	Move cursor one character to the right on your command line. If at the end of the line, it inserts a character from the text of your last command at that position.
F2	Create a new command line by copying your last command line up to the character that you type.
F3	Complete the command line with content from your last command line, from the current cursor position to the end.
F4	Delete characters from your cursor position up to (but not including) the character that you type.
F5	Scan backward through your command history.
F7	Interactively select a command from your command history. Use the arrow keys to scroll through the window that appears. Press the Enter key to execute the command, or use the right arrow key to place the text on your command line instead.
F8	Scan backward through your command history, only displaying matches for commands that match the text you've typed so far on the command line.
F9	Invoke a specific numbered command from your command history. The numbers of these commands correspond to the numbers that the command-history selection window (F7) shows.
Alt + F7	Clear the command history list.




While useful in their own right, the hotkeys listed in [Table 2-19](#) become even more useful when you map them to shorter or more intuitive keystrokes using a hotkey program, such as the free AutoHotkey <http://www.autohotkey.com>.

2.20.2. Profiles

Windows PowerShell automatically runs the four scripts listed in [Table 2-20](#) during startup. Each, if present, lets you customize your execution environment. PowerShell runs anything you place in these files as though you had entered it manually at the command line.

Table 2-20. Windows PowerShell profiles

Profile purpose	Profile location
Customization of all PowerShell sessions, including PowerShell hosting applications for all users on the system	<code>InstallationDirectory\profile.ps1</code>
Customization of <i>PowerShell.exe</i> sessions for all users on the system	<code>InstallationDirectory\Microsoft.PowerShell_profile.ps1</code>
Customization of all PowerShell sessions, including PowerShell hosting applications	<code>My Documents\WindowsPowerShell\profile.ps1</code>
Typical customization of <i>PowerShell.exe</i> sessions	<code>My Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1</code>



PowerShell makes editing your profile script simple by defining the automatic variable, `$profile`.

To create a new profile, type:

```
New-Item -Type file -Force $profile
```

To edit this profile, type:

```
Notepad $profile
```

For more information on writing scripts, see "Writing Scripts, Reusing Functionality," earlier in this chapter.

2.20.3. Prompts

To customize your prompt, add a "prompt" function to your profile. This function returns a string. For example:

```
function Prompt
{
    "PS [$env:COMPUTERNAME] >"
}
```

2.20.4. Tab Completion

You may define a `TabExpansion` function to customize the way that Windows PowerShell completes properties,

variables, parameters, and files when you press the Tab key.

Your `TabExpansion` function overrides the one that PowerShell defines by default, though, so you may want to use its definition as a starting point:

```
Get-Content function:\TabExpansion
```

As its arguments, this function receives the entire command line as input, as well as the last word of the command line. If the function returns one or more strings, PowerShell cycles through those strings during tab completion. Otherwise, it uses its built-in logic to tab-complete file names, directory names, cmdlet names, and variable names.





Chapter 3. Regular Expression Reference

Regular expressions play an important role in most text parsing and text matching tasks. They form an important underpinning of the `-match` operator, the `switch` statement, the `Select-String` cmdlet, and more. Tables Table 3-1 through Table 3-9 list commonly used regular expressions.

Table 3-1. Character classes: Patterns that represent sets of characters

Character class	Matches
.	Any character except for a newline. If the regular expression uses the <code>SingleLine</code> option, it matches any character. PS >"T" -match '.' True
<code>[characters]</code>	Any character in the brackets. For example: <code>[aeiou]</code> . PS >"Test" -match '[Tes]' True
<code>[^characters]</code>	Any character not in the brackets. For example: <code>[^aeiou]</code> . PS >"Test" -match '[^Tes]' False
<code>[start-end]</code>	Any character between the characters <code>start</code> and <code>end</code> , inclusive. You may include multiple character ranges between the brackets. For example, <code>[a-eh-j]</code> . PS >"Test" -match '[e-t]' True
<code>[^start-end]</code>	Any character not between any of the character ranges <code>start</code> through <code>end</code> , inclusive. You may include multiple character ranges between the brackets. For example, <code>[^a-eh-j]</code> . PS >"Test" -match '[^e-t]' False
<code>\p{character class}</code>	Any character in the Unicode group or block range specified by <code>{character class}</code> . PS >"+" -match '\p{Sm}' True
<code>\P{character class}</code>	Any character not in the Unicode group or block range specified by <code>{character class}</code> . PS >"+" -match '\P{Sm}' False
<code>\w</code>	Any word character. PS >"a" -match '\w' True

Character class	Matches
<code>\w</code>	Any nonword character. PS >"!" -match '\w' True
<code>\s</code>	Any whitespace character. PS >"`t" -match '\s' True
<code>\S</code>	Any nonwhitespace character. PS >" `t" -match '\S' False
<code>\d</code>	Any decimal digit. PS >"5" -match '\d' True
<code>\D</code>	Any nondecimal digit. PS >"!" -match '\D' True

Table 3-2. Quantifiers: Expressions that enforce quantity on the preceding expression

Quantifier	Meaning
<none>	One match. PS >"T" -match 'T' True
*	Zero or more matches, matching as much as possible. PS >"A" -match 'T*' True PS >"TTTTT" -match '^T*\$' True
+	One or more matches, matching as much as possible. PS >"A" -match 'T+' False PS >"TTTTT" -match '^T+\$' True

Quantifier	Meaning
?	Zero or one matches, matching as much as possible. PS >"TTTTT" -match '^T?\$' False
{ <i>n</i> }	Exactly <i>n</i> matches. PS >"TTTTT" -match '^T{5}\$' True
{ <i>n</i> ,}	<i>n</i> or more matches, matching as much as possible. PS >"TTTTT" -match '^T{4,}\$' True
{ <i>n</i> , <i>m</i> }	Between <i>n</i> and <i>m</i> matches (inclusive), matching as much as possible. PS >"TTTTT" -match '^T{4,6}\$' True
*?	Zero or more matches, matching as little as possible. PS >"A" -match '^AT *?\$', True
+?	One or more matches, matching as little as possible. PS >"A" -match '^AT +?\$', False
??	Zero or one matches, matching as little as possible. PS >"A" -match '^AT ??\$', True
{ <i>n</i> }?	Exactly <i>n</i> matches. PS >"TTTTT" -match '^T{5}?\$', True
{ <i>n</i> ,}?	<i>n</i> or more matches, matching as little as possible. PS >"TTTTT" -match '^T{4,}?\$', True
{ <i>n</i> , <i>m</i> }?	Between <i>n</i> and <i>m</i> matches (inclusive), matching as little as possible. PS >"TTTTT" -match '^T{4,6}?\$', True

Table 3-3. Grouping constructs: Expressions that let you group characters, patterns, and other expressions

Grouping construct	Description														
(<i>text</i>)	<p>Captures the text matched inside the parentheses. These captures are named by number (starting at one) based on the order of the opening parenthesis.</p> <pre>PS >"Hello" -match '^(.*)llo\$'; \$matches[1] True He</pre>														
(? <i>name</i>)	<p>Captures the text matched inside the parentheses. These captures are named by the name given in <i>name</i>.</p> <pre>PS >"Hello" -match '^(?<One>.*)llo\$'; \$matches.One True He</pre>														
(? <i>name1-name2</i>)	<p>A balancing group definition. This is an advanced regular expression construct, but allows you to match evenly balanced pairs of terms.</p>														
(?:)	<p>Noncapturing group.</p> <pre>PS >"A1" -match '((A B)\d)'; \$matches True</pre> <table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>2</td> <td>A</td> </tr> <tr> <td>1</td> <td>A1</td> </tr> <tr> <td>0</td> <td>A1</td> </tr> </tbody> </table> <pre>PS >"A1" -match '((?:A B)\d)'; \$matches True</pre> <table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>A1</td> </tr> <tr> <td>0</td> <td>A1</td> </tr> </tbody> </table>	Name	Value	2	A	1	A1	0	A1	Name	Value	1	A1	0	A1
Name	Value														
2	A														
1	A1														
0	A1														
Name	Value														
1	A1														
0	A1														
(? <i>imnsx-imnsx:</i>)	<p>Applies or disables the given option for this group. Supported options are:</p> <ul style="list-style-type: none"> <i>i</i> case-insensitive <i>m</i> multiline <i>n</i> explicit capture <i>s</i> single line <i>x</i> ignore whitespace <pre>PS >"Te`nst" -match '(T e.st)' False PS >"Te`nst" -match '(?sx:T e.st)'</pre>														

Grouping construct	Description
	True
(?=)	Zero-width positive lookahead assertion. Ensures that the given pattern matches to the right, without actually performing the match. <pre>PS >"555-1212" -match '(?=...-)(.*)'; \$matches[1] True 555-1212</pre>
(?!)	Zero-width negative lookahead assertion. Ensures that the given pattern does not match to the right, without actually performing the match. <pre>PS >"friendly" -match '(?!friendly)friend' False</pre>
(?<=)	Zero-width positive lookbehind assertion. Ensures that the given pattern matches to the left, without actually performing the match. <pre>PS >"public int X" -match '^.*(?<=public)int .*\$' True</pre>
(?<!)	Zero-width negative lookbehind assertion. Ensures that the given pattern does not match to the left, without actually performing the match. <pre>PS >"private int X" -match '^.*(?<!private)int .*\$' False</pre>
(?>)	Nonbacktracking subexpression. Matches only if this subexpression can be matched completely. <pre>PS >"Hello World" -match '(Hello.*)orld' True PS >"Hello World" -match '(?>Hello.*)orld' False</pre> <p>The nonbacktracking version of the subexpression fails to match, as its complete match would be "Hello World".</p>

Table 3-4. Atomic zero-width assertions: Patterns that restrict where a match may occur

Assertion	Restriction
-----------	-------------

Assertion	Restriction
^	The match must occur at the beginning of the string (or line, if the <code>Multiline</code> option is in effect). <pre>PS >"Test" -match '^est'</pre> <pre>False</pre>
\$	The match must occur at the end of the string (or line, if the <code>Multiline</code> option is in effect). <pre>PS >"Test" -match 'Tes\$'</pre> <pre>False</pre>
\A	The match must occur at the beginning of the string. <pre>PS >"The`nTest" -match '(?m:^Test)'</pre> <pre>True</pre> <pre>PS >"The`nTest" -match '(?m:\ATest)'</pre> <pre>False</pre>
\Z	The match must occur at the end of the string or before <code>\n</code> at the end of the string. <pre>PS >"The`nTest`n" -match '(?m:The\$)'</pre> <pre>True</pre> <pre>PS >"The`nTest`n" -match '(?m:The \Z)'</pre> <pre>False</pre> <pre>PS >"The`nTest`n" -match 'Test\Z'</pre> <pre>True</pre>
\z	The match must occur at the end of the string. <pre>PS >"The`nTest`n" -match 'Test\z'</pre> <pre>False</pre>
\G	The match must occur where the previous match ended. Used with the <code>System.Text.RegularExpressions.Match.NextMatch()</code> method.
\b	The match must occur on a word boundary—the first or last characters in words separated by nonalphanumeric characters. <pre>PS >"Testing" -match 'ing\b'</pre> <pre>True</pre>
\B	The match must not occur on a word boundary. <pre>PS >"Testing" -match 'ing\B'</pre> <pre>False</pre>

Table 3-5. Substitution patterns: Patterns used in a regular expression-replace operation

Pattern	Substitution
---------	--------------

Pattern	Substitution
<code>\$number</code>	The text matched by group number <code><number></code> . <pre>PS >"Test" -replace '(.*)st','\$lar'</pre> Tear
<code>\${name}</code>	The text matched by group named <code><name></code> . <pre>PS >"Test" -replace '(?<pre>.*)st','\${pre}ar'</pre> Tear
<code>\$\$</code>	A literal <code>\$</code> . <pre>PS >"Test" -replace '.', '\$\$'</pre> \$\$\$\$
<code>\$&</code>	A copy of the entire match. <pre>PS >"Test" -replace '^.*\$', 'Found: \$&'</pre> Found: Test
<code>\$`</code>	The text of the input string that precedes the match. <pre>PS >"Test" -replace 'est\$', 'Te\$`'</pre> TTeT
<code>\$'</code>	The text of the input string that follows the match. <pre>PS >"Test" -replace '^Tes', 'Res\$'''</pre> Restt
<code>\$+</code>	The last group captured. <pre>PS >"Testing" -replace '(.*)ing','\$+ed'</pre> Tested
<code>\$_</code>	The entire input string. <pre>PS >"Testing" -replace '(.*)ing','String: \$_'</pre> String: Testing

Table 3-6. Alternation constructs: Expressions that allow you to perform either/or logic

Alternation construct	Description
<code> </code>	Matches any of the terms separated by the vertical bar character. <pre>PS >"Test" -match '(B T)est'</pre> True
<code>(?(expression) yes no)</code>	Matches the <i>yes</i> term if expression matches at this point. Otherwise, matches the <i>no</i> term. The <i>no</i> term is optional.

Alternation construct	Description
	<pre>PS >"3.14" -match '(?(\d)3.14 Pi)' True PS >"Pi" -match '(?(\d)3.14 Pi)' True PS >"2.71" -match '(?(\d)3.14 Pi)' False</pre>
(? <i>name</i>) <i>yes</i> <i>no</i>)	<p>Matches the <i>yes</i> term if the capture group named <i>name</i> has a capture at this point. Otherwise, matches the <i>no</i> term. The <i>no</i> term is optional.</p> <pre>PS >"123" -match '(?<one>1)?(? (one) 23 234) ' True PS >"23" -match '(?<one>1)?(? (one) 23 234) ' False PS >"234" -match '(?<one>1)?(? (one) 23 234) ' True</pre>

Table 3-7. Backreference constructs: Expressions that refer to a capture group within the expression

Backreference construct	Refers to
<code>\number</code>	<p>Group number <i>number</i> in the expression.</p> <pre>PS >" Text " -match '(.)Text\1' True PS >" Text+" -match '(.)Text\1' False</pre>
<code>\k<name></code>	<p>The group named <i>name</i> in the expression.</p> <pre>PS >" Text " -match '(?<Symbol>.)Text\k<Symbol>' True PS >" Text+" -match '(?<Symbol>.)Text\k<Symbol>' False</pre>

Table 3-8. Other constructs: Other expressions that modify a regular expression

Construct	Description
(?imnsx-imnsx)	Applies or disables the given option for the rest of this expression. Supported options are: <i>i</i> case-insensitive <i>m</i> multiline <i>n</i> explicit capture <i>s</i> single line <i>x</i> ignore whitespace PS >"Te`nst" -match '(?sx)T e.st' True
(?#)	Inline comment. This terminates at the first closing parenthesis. PS >"Test" -match '(?# Match 'Test')Test' True
# [to end of line]	Comment form allowed when the regular expression has the <code>IgnoreWhitespace</code> option enabled. PS >"Test" -match '(?x)Test # Matches Test' True

Table 3-9. Character escapes: Character sequences that represent another character

Escaped character	Match
<ordinary characters>	Characters other than . \$ ^ { [() * + ? \ match themselves.
\a	A bell (alarm) \u0007.
\b	A backspace \u0008 if in a [] character class. In a regular expression, \b denotes a word boundary (between \w and \W characters) except within a [] character class, where \b refers to the backspace character. In a replacement pattern, \b always denotes a backspace.
\t	A tab \u0009.
\r	A carriage return \u000D.
\v	A vertical tab \u000B.
\f	A form feed \u000C.
\n	A new line \u000A.
\e	An escape \u001B.

Escaped character	Match
<code>\ddd</code>	An ASCII character as octal (up to three digits). Numbers with no leading zero are treated as backreferences if they have only one digit, or if they correspond to a capturing group number.
<code>\xdd</code>	An ASCII character using hexadecimal representation (exactly two digits).
<code>\cC</code>	An ASCII control character. For example, <code>\cC</code> is Control-C.
<code>\udddd</code>	A Unicode character using hexadecimal representation (exactly four digits).
<code>\</code>	When followed by a character that is not recognized as an escaped character, matches that character. For example, <code>*</code> is the literal character <code>*</code> .





Chapter 4. PowerShell Automatic Variables

PowerShell defines and populates several variables automatically. These variables let you access information about the execution environment, PowerShell preferences, and more.

Table 4-1 provides a listing of these automatic variables and their meanings.

Table 4-1. Windows PowerShell automatic variables: Variables automatically used and set by Windows PowerShell

Variable	Meaning
<code>\$\$</code>	Last token of the last line received by the shell.
<code>\$?</code>	Success/fail status of the last operation.
<code>\$^</code>	First token of the last line received by the shell.
<code>\$_</code>	Current pipeline object in a pipelined script block.
<code>\$args</code>	Array of parameters passed to the script, function, or script block.
<code>\$confirmPreference</code>	Preference that controls the level of impact that operations may have before requesting confirmation. Supports the values <code>none</code> , <code>low</code> , <code>medium</code> , <code>high</code> . A value of <code>none</code> disables confirmation messages.
<code>\$consoleFilename</code>	Filename of the PowerShell console file that configured this session, if one was used.
<code>\$currentlyExecutingCommand</code>	Currently executing command, when in a suspended prompt.
<code>\$debugPreference</code>	Preference that controls how PowerShell should handle debug output written by a script or cmdlet. Supports the values <code>SilentlyContinue</code> , <code>Continue</code> , <code>Inquire</code> , and <code>Stop</code> .
<code>\$error</code>	Array that holds the terminating and nonterminating errors generated in the shell.
<code>\$errorActionPreference</code>	Preference that controls how PowerShell should handle error output written by a script or cmdlet. Supports the values <code>SilentlyContinue</code> , <code>Continue</code> , <code>Inquire</code> , and <code>Stop</code> .
<code>\$errorView</code>	Preference that controls how PowerShell should output errors in the shell. Supports the values of <code>Normal</code> and <code>CategoryView</code> (a more succinct and categorical view of the error).
<code>\$executionContext</code>	Means by which scripts can access the APIs typically used by cmdlets and providers.
<code>\$false</code>	Variable that represents the Boolean value <code>False</code> .
<code>\$foreach</code>	Enumerator within a <code>foreach</code> loop.
<code>\$formatEnumerationLimit</code>	Limit on how deep into an object the formatting and output facilities travel before outputting an object.

Variable	Meaning
<code>\$home</code>	User's home directory.
<code>\$host</code>	Means by which scripts can access the APIs and implementation details of the current host and user interface.
<code>\$input</code>	Current input pipeline in a pipelined script block.
<code>\$lastExitCode</code>	Exit code of the last command. Can be explicitly set by scripts, and is automatically set when calling native executables.
<code>\$logEngineHealthEvent</code>	Preference that tells PowerShell to log engine health events, such as errors and exceptions. Supports the values <code>\$true</code> and <code>\$false</code> .
<code>\$logEngineLifecycleEvent</code>	Preference that tells PowerShell to log engine lifecycle events, such as <code>Start</code> and <code>Stop</code> . Supports the values <code>\$true</code> and <code>\$false</code> .
<code>\$logCommandHealthEvent</code>	Preference that tells PowerShell to log command health events, such as errors and exceptions. Supports the values <code>\$true</code> and <code>\$false</code> .
<code>\$logCommandLifecycleEvent</code>	Preference that tells PowerShell to log command lifecycle events, such as <code>Start</code> and <code>Stop</code> . Supports the values <code>\$true</code> and <code>\$false</code> .
<code>\$logProviderHealthEvent</code>	Preference that tells PowerShell to log provider health events, such as errors and exceptions. Supports the values <code>\$true</code> and <code>\$false</code> .
<code>\$logProviderLifecycleEvent</code>	Preference that tells PowerShell to log provider lifecycle events, such as <code>Start</code> and <code>Stop</code> . Supports the values <code>\$true</code> and <code>\$false</code> .
<code>\$matches</code>	Results of the last successful regular expression match (through the <code>-match</code> operator).
<code>\$maximumAliasCount</code>	Limit on how many aliases may be defined.
<code>\$maximumDriveCount</code>	Limit on how many drives may be defined. Does not include default system drives.
<code>\$maximumErrorCount</code>	Limit on how many errors PowerShell retains in the <code>\$error</code> collection.
<code>\$maximumFunctionCount</code>	Limit on how many functions may be defined.
<code>\$maximumHistoryCount</code>	Limit on how many history items are retained.
<code>\$maximumVariableCount</code>	Limit on how many variables may be defined.
<code>\$myInvocation</code>	Information about the context under which the script, function, or script block was run, including detailed information about the command (<code>MyCommand</code>) and the script that defines it (<code>ScriptName</code>).
<code>\$nestedPromptLevel</code>	Nesting level of the current prompt. Incremented by operations that enter a nested prompt (such as <code>\$host.EnterNestedPrompt()</code>) and decremented by the <code>exit</code> statement.
<code>\$null</code>	Variable that represents the concept of <code>Null</code> .
<code>\$ofs</code>	Output field separator. Placed between elements when PowerShell outputs a list as a string.
<code>\$outputEncoding</code>	Character encoding used when sending pipeline data to external processes.

Variable	Meaning
<code>\$pid</code>	Process ID of the current PowerShell instance.
<code>\$profile</code>	Location and filename of the PowerShell profile for this host.
<code>\$progressPreference</code>	Preference that controls how PowerShell should handle progress output written by a script or cmdlet. Supports the values <code>SilentlyContinue</code> , <code>Continue</code> , <code>Inquire</code> , and <code>Stop</code> .
<code>\$psHome</code>	Installation location of PowerShell.
<code>\$pwd</code>	Current working directory.
<code>\$shellId</code>	Shell identifier of this host.
<code>\$stackTrace</code>	Detailed stack trace information of the last error.
<code>\$this</code>	Reference to the current object in <code>ScriptMethods</code> and <code>ScriptProperties</code> .
<code>\$transcript</code>	Filename used by the <code>Start-Transcript</code> cmdlet.
<code>\$true</code>	Variable that represents the Boolean value <code>True</code> .
<code>\$verboseHelpErrors</code>	Preference that tells PowerShell to output detailed error information when parsing malformed help files. Supports the values <code>\$true</code> and <code>\$false</code> .
<code>\$verbosePreference</code>	Preference that controls how PowerShell should handle verbose output written by a script or cmdlet. Supports the values <code>SilentlyContinue</code> , <code>Continue</code> , <code>Inquire</code> , and <code>Stop</code> .
<code>\$warningPreference</code>	Preference that controls how PowerShell should handle warning output written by a script or cmdlet. Supports the values <code>SilentlyContinue</code> , <code>Continue</code> , <code>Inquire</code> , and <code>Stop</code> .
<code>\$whatifPreference</code>	Preference that controls how PowerShell should handle confirmation requests called by a script or cmdlet. Supports the values <code>SilentlyContinue</code> , <code>Continue</code> , <code>Inquire</code> , and <code>Stop</code> .



Chapter 5. Standard PowerShell Verbs

Cmdlets and scripts should be named using a *Verb-Noun* syntax, for example, `Get-ChildItem`. The official guidance is that, with rare exception, cmdlets should use the standard PowerShell verbs. They should avoid any synonyms or concepts that can be mapped to the standard. This allows administrators to quickly understand a set of cmdlets that use a new noun.

Verbs should be phrased in the present tense, and nouns should be singular. Tables [Table 5-1](#) through [Table 5-6](#) list the different categories of standard PowerShell verbs.

Table 5-1. Standard Windows PowerShell common verbs

Verb	Meaning	Synonyms
Add	Adds a resource to a container, or attaches an element to another element.	Append, Attach, Concatenate, Insert
Clear	Removes all elements from a container.	Flush, Erase, Release, Unmark, Unset, Nullify
Copy	Copies a resource to another name or container.	Duplicate, Clone, Replicate
Get	Retrieves data.	Read, Open, Cat, Type, Dir, Obtain, Dump, Acquire, Examine, Find, Search
Hide	Makes a display not visible.	Suppress
Join	Joins a resource.	Combine, Unite, Connect, Associate
Lock	Locks a resource.	Restrict, Bar
Move	Moves a resource.	Transfer, Name, Migrate
New	Creates a new resource.	Create, Generate, Build, Make, Allocate
Push	Puts an item onto the top of a stack.	Put, Add, Copy
Pop	Removes an item from the top of a stack.	Remove, Paste
Remove	Removes a resource from a container.	Delete, Kill
Rename	Gives a resource a new name.	Ren, Swap
Search	Finds a resource (or summary information about that resource) in a collection. Does not actually retrieve the resource, but provides information to be used when retrieving it.	Find, Get, Grep, Select
Select	Creates a subset of data from a larger data set.	Pick, Grep, Filter
Set	Places data.	Write, Assign, Configure
Show	Retrieves, formats, and displays information.	Display, Report

Verb	Meaning	Synonyms
Split	Separates data into smaller elements.	Divide, Chop, Parse
Unlock	Unlocks a resource.	Free, Unrestrict
Use	Applies or associates a resource with a context.	With, Having

Table 5-2. Standard Windows PowerShell communication verbs

Verb	Meaning	Synonyms
Connect	Connects a source to a destination.	Join, Telnet
Disconnect	Disconnects a source from a destination.	Break, Logoff
Read	Acquires information from a nonconnected source.	Prompt, Get
Receive	Acquires information from a connected source.	Read, Accept, Peek
Send	Writes information to a connected destination.	Put, Broadcast, Mail
Write	Writes information to a nonconnected destination.	Put, Print

Table 5-3. Standard Windows PowerShell data verbs

Verb	Meaning	Synonyms
Backup	Backs up data.	Save, Burn
Checkpoint	Creates a snapshot of the current state of data or its configuration.	Diff, StartTransaction
Compare	Compares a resource with another resource.	Diff, Bc
Convert	Changes from one representation to another, when the cmdlet supports bidirectional conversion, or conversion of many data types.	Change, Resize, Resample
ConvertFrom	Converts from one primary input to several supported outputs.	Export, Output, Out
ConvertTo	Converts from several supported inputs to one primary output.	Import, Input, In
Dismount	Detaches a name entity from a location in a namespace.	Dismount, Unlink
Export	Stores the primary input resource into a backing store or interchange format.	Extract, Backup
Import	Creates a primary output resource from a backing store or interchange format.	Load, Read
Initialize	Prepares a resource for use, and initializes it to a default state.	Setup, Renew, Rebuild
Limit	Applies constraints to a resource.	Quota, Enforce
Merge	Creates a single data instance from multiple data sets.	Combine, Join

Verb	Meaning	Synonyms
Mount	Attaches a named entity to a location in a namespace.	Attach, Link
Out	Sends data to a terminal location.	Print, Format, Send
Publish	Make a resource known or visible to others.	Deploy, Release, Install
Restore	Restores a resource to a set of conditions that have been predefined or set by a checkpoint.	Repair, Return, Fix
Unpublish	Removes a resource from public visibility.	Uninstall, Revert
Update	Updates or refreshes a resource.	Refresh, Renew, Index

Table 5-4. Standard Windows PowerShell diagnostic verbs

Verb	Meaning	Synonyms
Debug	Examines a resource, diagnoses operational problems.	Attach, Diagnose
Measure	Identifies resources consumed by an operation, or retrieves statistics about a resource.	Calculate, Determine, Analyze
Ping	Determines whether a resource is active and responsive; in most instances, this should be replaced by the verb, Test .	Connect, Debug
Resolve	Maps a shorthand representation to a more complete one.	Expand, Determine
Test	Verifies the validity or consistency of a resource.	Diagnose, Verify, Analyze
Trace	Follows the activities of the resource.	Inspect, Dig

Table 5-5. Standard Windows PowerShell lifecycle verbs

Verb	Meaning	Synonyms
Disable	Configures an item to be unavailable.	Halt, Hide
Enable	Configures an item to be available.	Allow, Permit
Install	Places a resource in the specified location and optionally initializes it.	Setup, Configure
Installw	Calls or launches an activity that cannot be stopped.	Run, Call, Perform
Restart	Stops an operation and starts it again.	Recycle, Hup
Resume	Begins an operation after it has been suspended.	Continue
Start	Begins an activity.	Launch, Initiate
Stop	Discontinues an activity.	Halt, End, Discontinue

Verb	Meaning	Synonyms
<code>Suspend</code>	Pauses an operation, but does not discontinue it.	Pause, Sleep, Break
<code>Uninstall</code>	Removes a resource from the specified location.	Remove, Clear, Clean
<code>Wait</code>	Pauses until an expected event occurs.	Sleep, Pause, Join

Table 5-6. Standard Windows PowerShell security verbs

Verb	Meaning	Synonyms
<code>Block</code>	Restricts access to a resource.	Prevent, Limit, Deny
<code>Grant</code>	Grants access to a resource.	Allow, Enable
<code>Revoke</code>	Removes access to a resource.	Remove, Disable
<code>Unblock</code>	Removes a restriction of access to a resource.	Clear, Allow



Chapter 6. Selected .NET Classes and Their Uses

Tables Table 6-1 through Table 6-16 provide pointers to types in the .NET Framework that usefully complement the functionality that PowerShell provides. For detailed descriptions and documentation, search the official documentation at <http://msdn.microsoft.com>.

Table 6-1. Windows PowerShell

Class	Description
<code>System.Management.Automation.PSObject</code>	Represents a PowerShell object to which you can add notes, properties, and more.

Table 6-2. Utility

Class	Description
<code>System.DateTime</code>	Represents an instant in time, typically expressed as a date and time of day.
<code>System.Guid</code>	Represents a globally unique identifier (GUID).
<code>System.Math</code>	Provides constants and static methods for trigonometric, logarithmic, and other common mathematical functions.
<code>System.Random</code>	Represents a pseudorandom number generator, a device that produces a sequence of numbers that meet certain statistical requirements for randomness.
<code>System.Convert</code>	Converts a base data type to another base data type.
<code>System.Environment</code>	Provides information about, and means to manipulate, the current environment and platform.
<code>System.Console</code>	Represents the standard input, output, and error streams for console applications.
<code>System.Text.RegularExpressions.Regex</code>	Represents an immutable regular expression.
<code>System.Diagnostics.Debug</code>	Provides a set of methods and properties that help debug your code.
<code>System.Diagnostics.EventLog</code>	Provides interaction with Windows event logs.
<code>System.Diagnostics.Process</code>	Provides access to local and remote processes, and enables you to start and stop local system processes.
<code>System.Diagnostics.Stopwatch</code>	Provides a set of methods and properties that you can use to accurately measure elapsed time.
<code>System.Media.SoundPlayer</code>	Controls playback of a sound from a <code>.wav</code> file.

Table 6-3. Collections and object utilities

Class	Description
<code>System.Array</code>	Provides methods for creating, manipulating, searching, and sorting arrays, thereby serving as the base class for all arrays in the Common Language Runtime.
<code>System.Enum</code>	Provides the base class for enumerations.
<code>System.String</code>	Represents text as a series of Unicode characters.
<code>System.Text.StringBuilder</code>	Represents a mutable string of characters.
<code>System.Collections.Specialized.OrderedDictionary</code>	Represents a collection of key/value pairs that are accessible by the key or index.
<code>System.Collections.ArrayList</code>	Implements the <code>ICollection</code> interface using an array whose size is dynamically increased as required.

Table 6-4. The .NET Framework

Class	Description
<code>System.AppDomain</code>	Represents an application domain, which is an isolated environment where applications execute.
<code>System.Reflection.Assembly</code>	Defines an assembly, which is a reusable, versionable, and self-describing building block of a Common Language Runtime application.
<code>System.Type</code>	Represents type declarations: class types, interface types, array types, value types, enumeration types, type parameters, generic type definitions, and open or closed constructed generic types.
<code>System.Threading.Thread</code>	Creates and controls a thread, sets its priority, and gets its status.
<code>System.Runtime.InteropServices.Marshal</code>	Provides a collection of methods for allocating unmanaged memory, copying unmanaged memory blocks, and converting managed to unmanaged types, as well as other miscellaneous methods used when interacting with unmanaged code.
<code>Microsoft.CSharp.CSharpCodeProvider</code>	Provides access to instances of the C# code generator and code compiler.

Table 6-5. Registry

Class	Description
<code>Microsoft.Win32.Registry</code>	Provides <code>RegistryKey</code> objects that represent the root keys in the Windows registry, and static methods to access key/ value pairs.
<code>Microsoft.Win32.RegistryKey</code>	Represents a key-level node in the Windows registry.

Table 6-6. Input and output

Class	Description
<code>System.IO.Stream</code>	Provides a generic view of a sequence of bytes.
<code>System.IO.BinaryReader</code>	Reads primitive data types as binary values.
<code>System.IO.BinaryWriter</code>	Writes primitive types in binary to a stream.
<code>System.IO.BufferedStream</code>	Adds a buffering layer to read and write operations on another stream.
<code>System.IO.Directory</code>	Exposes static methods for creating, moving, and enumerating through directories and subdirectories.
<code>System.IO.FileInfo</code>	Provides instance methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of <code>FileStream</code> objects.
<code>System.IO.DirectoryInfo</code>	Exposes instance methods for creating, moving, and enumerating through directories and subdirectories.
<code>System.IO.File</code>	Provides static methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of <code>FileStream</code> objects.
<code>System.IO.MemoryStream</code>	Creates a stream whose backing store is memory.
<code>System.IO.Path</code>	Performs operations on string instances that contain file or directory path information. These operations are performed in a cross-platform manner.
<code>System.IO.TextReader</code>	Represents a reader that can read a sequential series of characters.
<code>System.IO.StreamReader</code>	Implements a <code>TextReader</code> that reads characters from a byte stream in a particular encoding.
<code>System.IO.TextWriter</code>	Represents a writer that can write a sequential series of characters.
<code>System.IO.StreamWriter</code>	Implements a <code>TextWriter</code> for writing characters to a stream in a particular encoding.
<code>System.IO.StringReader</code>	Implements a <code>TextReader</code> that reads from a string.
<code>System.IO.StringWriter</code>	Implements a <code>TextWriter</code> for writing information to a string.
<code>System.IO.Compression.DeflateStream</code>	Provides methods and properties used to compress and decompress streams using the Deflate algorithm.
<code>System.IO.Compression.GZipStream</code>	Provides methods and properties used to compress and decompress streams using the GZip algorithm.
<code>System.IO.FileSystemWatcher</code>	Listens to the file system change notifications and raises events when a directory, or file in a directory, changes.

Table 6-7. Security

Class	Description
<code>System.Security.Principal.WindowsIdentity</code>	Represents a Windows user.

Class	Description
<code>System.Security.Principal.WindowsPrincipal</code>	Allows code to check the Windows group membership of a Windows user.
<code>System.Security.Principal.WellKnownSidType</code>	Defines a set of commonly used security identifiers (SIDs).
<code>System.Security.Principal.WindowsBuiltInRole</code>	Specifies common roles to be used with <code>IsInRole</code> .
<code>System.Security.SecureString</code>	Represents text that should be kept confidential. The text is encrypted for privacy when being used and deleted from computer memory when no longer needed.
<code>System.Security.Cryptography.TripleDESCryptoServiceProvider</code>	Defines a wrapper object to access the cryptographic service provider (CSP) version of the TripleDES algorithm.
<code>System.Security.Cryptography.PasswordDeriveBytes</code>	Derives a key from a password using an extension of the PBKDF1 algorithm.
<code>System.Security.Cryptography.SHA1</code>	Computes the SHA1 hash for the input data.
<code>System.Security.AccessControl.FileSystemSecurity</code>	Represents the access control and audit security for a file or directory.
<code>System.Security.AccessControl.RegistrySecurity</code>	Represents the Windows access control security for a registry key.

Table 6-8. User interface

Class	Description
<code>System.Windows.Forms.Form</code>	Represents a window or dialog box that makes up an application's user interface.
<code>System.Windows.Forms.FlowLayoutPanel</code>	Represents a panel that dynamically lays out its contents.

Table 6-9. Image manipulation

Class	Description
<code>System.Drawing.Image</code>	A class that provides functionality for the <code>Bitmap</code> and <code>Metafile</code> classes.
<code>System.Drawing.Bitmap</code>	Encapsulates a GDI+ bitmap, which consists of the pixel data for a graphics image and its attributes. A bitmap is an object used to work with images defined by pixel data.

Table 6-10. Networking

Class	Description
-------	-------------

Class	Description
<code>System.Uri</code>	Provides an object representation of a uniform resource identifier (URI) and easy access to the parts of the URI.
<code>System.Net.NetworkCredential</code>	Provides credentials for password-based authentication schemes, such as basic, digest, NTLM, and Kerberos authentication.
<code>System.Net.Dns</code>	Provides simple domain name resolution functionality.
<code>System.Net.FtpWebRequest</code>	Implements a File Transfer Protocol (FTP) client.
<code>System.Net.HttpWebRequest</code>	Provides an HTTP-specific implementation of the <code>WebRequest</code> class.
<code>System.Net.WebClient</code>	Provides common methods for sending data to and receiving data from a resource identified by a URI.
<code>System.Net.Sockets.TcpClient</code>	Provides client connections for TCP network services.
<code>System.Net.Mail.MailAddress</code>	Represents the address of an electronic mail sender or recipient.
<code>System.Net.Mail.MailMessage</code>	Represents an email message that can be sent using the <code>SmtpClient</code> class.
<code>System.Net.Mail.SmtpClient</code>	Allows applications to send email by using the Simple Mail Transfer Protocol (SMTP).
<code>System.IO.Ports.SerialPort</code>	Represents a serial port resource.
<code>System.Web.HttpUtility</code>	Provides methods for encoding and decoding URLs when processing web requests.

Table 6-11. XML

Class	Description
<code>System.Xml.XmlTextWriter</code>	Represents a writer that provides a fast, noncached, forward-only way of generating streams or files containing XML data that conforms to the W3C Extensible Markup Language (XML) 1.0 and the namespaces in XML recommendations.
<code>System.Xml.XmlDocument</code>	Represents an XML document.

Table 6-12. Windows Management Instrumentation

Class	Description
<code>System.Management.ManagementObject</code>	Represents a WMI instance.
<code>System.Management.ManagementClass</code>	Represents a management class. A management class is a WMI class, such as <code>Win32_LogicalDisk</code> , which can represent a disk drive, and <code>Win32_Process</code> , which represents a process, such as an instance of <i>Notepad.exe</i> . The members of this class enable you to access WMI data using a specific WMI class path. For more information, see "Win32 Classes" in the Windows Management Instrumentation documentation in the MSDN Library at http://msdn.microsoft.com/library .

Class	Description
<code>System.Management.ManagementObjectSearcher</code>	Retrieves a collection of WMI management objects based on a specified query. This class is one of the more commonly used entry points to retrieving management information. For example, it can be used to enumerate all disk drives, network adapters, processes, and many more management objects on a system, or to query for all network connections that are up, services that are paused, and so on. When instantiated, an instance of this class takes as input a WMI query, represented in an <code>ObjectQuery</code> or its derivatives, and optionally a <code>ManagementScope</code> , representing the WMI namespace to execute the query in. It can also take additional advanced options in an <code>EnumerationOptions</code> . When the <code>Get</code> method on this object is invoked, the <code>ManagementObjectSearcher</code> executes the given query in the specified scope and returns a collection of management objects that match the query in a <code>ManagementObjectCollection</code> .
<code>System.Management.ManagementDateTimeConverter</code>	Provides methods to convert DMTF datetime and time intervals to CLR-compliant <code>DateTime</code> and <code>TimeSpan</code> formats and vice versa.
<code>System.Management.ManagementEventWatcher</code>	Subscribes to temporary event notifications based on a specified event query.

Table 6-13. Active Directory

Class	Description
<code>System.DirectoryServices.DirectorySearcher</code>	Performs queries against Active Directory.
<code>System.DirectoryServices.DirectoryEntry</code>	The <code>DirectoryEntry</code> class encapsulates a node or object in the Active Directory hierarchy.

Table 6-14. Database

Class	Description
<code>System.Data.DataSet</code>	Represents an in-memory cache of data.
<code>System.Data.DataTable</code>	Represents one table of in-memory data.
<code>System.Data.SqlClient.SqlCommand</code>	Represents a <code>Transact-SQL</code> statement or stored procedure to execute against a SQL Server database.
<code>System.Data.SqlClient.SqlConnection</code>	Represents an open connection to a SQL Server database.
<code>System.Data.SqlClient.SqlDataAdapter</code>	Represents a set of data commands and a database connection that are used to fill the <code>DataSet</code> and update a SQL Server database.

Class	Description
<code>System.Data.Odbc.OdbcCommand</code>	Represents a SQL statement or stored procedure to execute against a data source.
<code>System.Data.Odbc.OdbcConnection</code>	Represents an open connection to a data source.
<code>System.Data.Odbc.OdbcDataAdapter</code>	Represents a set of data commands and a connection to a data source that are used to fill the <code>DataSet</code> and update the data source.

Table 6-15. Message queuing

Class	Description
<code>System.Messaging.MessageQueue</code>	Provides access to a queue on a Message Queuing server.

Table 6-16. Transactions

Class	Description
<code>System.Transactions.Transaction</code>	Represents a transaction.





Chapter 7. WMI Reference

The Windows Management Instrumentation (WMI) facilities in Windows offer thousands of classes that provide information of interest to administrators. [Table 7-1](#) lists the categories and subcategories covered by WMI and can be used to get a general idea of the scope of WMI classes. [Table 7-2](#) provides a selected subset of the most useful WMI classes. For more information about a category, search the official WMI documentation at <http://msdn.microsoft.com>.

Table 7-1. WMI class categories and subcategories

Category	Subcategory
Computer System Hardware	Cooling device, input device, mass storage, motherboard, controller and port, networking device, power, printing, telephony, video, and monitor
Operating System	COM, desktop, drivers, filesystem, job objects, memory and page files, multimedia audio/visual, networking, operating system events, operating system settings, processes, registry, scheduler jobs, security, services, shares, Start menu, storage, users, Windows NT event log, Windows product activation
WMI Service Management	WMI configuration, WMI management
General	Installed applications, performance counter, security descriptor

Table 7-2. Selected WMI classes

Class	Description
Win32_BaseBoard	Represents a baseboard, which is also known as a motherboard or system board.
Win32_BIOS	Represents the attributes of the computer system's basic input/output services (BIOS) that are installed on a computer.
Win32_BootConfiguration	Represents the boot configuration of a Windows system.
Win32_CDROMDrive	Represents a CD-ROM drive on a Windows computer system. Be aware that the name of the drive does not correspond to the logical drive letter assigned to the device.
Win32_ComputerSystem	Represents a computer system in a Windows environment.
Win32_Processor	Represents a device that can interpret a sequence of instructions on a computer running on a Windows operating system. On a multiprocessor computer, one instance of the Win32_Processor class exists for each processor.
Win32_ComputerSystemProduct	Represents a product. This includes software and hardware used on this computer system.
CIM_DataFile	Represents a named collection of data or executable code. Currently, the provider returns files on fixed and mapped logical disks. In the future, only instances of files on local fixed disks will be returned.

Class	Description
<code>Win32_DCOMApplication</code>	Represents the properties of a DCOM application.
<code>Win32_Desktop</code>	Represents the common characteristics of a user's desktop. The properties of this class can be modified by the user to customize the desktop.
<code>Win32_DesktopMonitor</code>	Represents the type of monitor or display device attached to the computer system.
<code>Win32_DeviceMemoryAddress</code>	Represents a device memory address on a Windows system.
<code>Win32_DiskDrive</code>	Represents a physical disk drive as seen by a computer running the Windows operating system. Any interface to a Windows physical disk drive is a descendant (or member) of this class. The features of the disk drive seen through this object correspond to the logical and management characteristics of the drive. In some cases, this may not reflect the actual physical characteristics of the device. Any object based on another logical device would not be a member of this class.
<code>Win32_DiskQuota</code>	Tracks disk space usage for NTFS filesystem volumes. A system administrator (SA) can configure Windows to prevent further disk space use and log an event when a user exceeds a specified disk space limit. An SA can also log an event when a user exceeds a specified disk space warning level. This class is new in Windows XP.
<code>Win32_DMACHannel</code>	Represents a direct memory access (DMA) channel on a Windows computer system. DMA is a method of moving data from a device to memory (or vice versa) without the help of the microprocessor. The system board uses a DMA controller to handle a fixed number of channels, each of which can be used by one (and only one) device at a time.
<code>Win32_Environment</code>	Represents an environment or system environment setting on a Windows computer system. Querying this class returns environment variables found in: <code>HKLM\System\CurrentControlSet\Control\Sessionmanager\Environment</code> as well as: <code>HKEY_USERS\<user sid="">\Environment</user></code>
<code>Win32_Directory</code>	Represents a directory entry on a Windows computer system. A <i>directory</i> is a type of file that logically groups data files and provides path information for the grouped files. <code>Win32_Directory</code> does not include directories of network drives.
<code>Win32_Group</code>	Represents data about a group account. A group account allows access privileges to be changed for a list of users, for example, administrators.
<code>Win32_IDEController</code>	Manages the capabilities of an integrated device electronics (IDE) controller device.

Class	Description
Win32_IRQResource	Represents an interrupt request line (IRQ) number on a Windows computer system. An interrupt request is a signal sent to the CPU by a device or program for time-critical events. IRQ can be hardware- or software-based.
Win32_ScheduledJob	<p>Represents a job created with the AT command. The Win32_ScheduledJob class does not represent a job created with the Scheduled Task Wizard from the Control Panel. You cannot change a task created by WMI in the Scheduled Tasks UI. Windows 2000 and Windows NT 4.0: You can use the Scheduled Tasks UI to modify the task you originally created with WMI. However, although the task is successfully modified, you can no longer access the task using WMI.</p> <p>Each job scheduled against the schedule service is stored persistently (the scheduler can start a job after a reboot) and is executed at the specified time and day of the week or month. If the computer is not active or if the scheduled service is not running at the specified job time, the schedule service runs the specified job on the next day at the specified time.</p> <p>Jobs are scheduled according to Universal Coordinated Time (UTC) with bias offset from Greenwich mean time (GMT), which means that a job can be specified using any time zone. The Win32_ScheduledJob class returns the local time with UTC offset when enumerating an object and converts to local time when creating new jobs. For example, a job specified to run on a computer in Boston at 10:30 P.M. Monday PST will be scheduled to run locally at 1:30 A.M. Tuesday EST. Note that a client must take into account whether Daylight Savings Time is in operation on the local computer, and if it is, then subtract a bias of 60 minutes from the UTC offset.</p>
Win32_LoadOrderGroup	Represents a group of system services that define execution dependencies. The services must be initiated in the order specified by the Load Order Group as the services are dependent on each other. These dependent services require the presence of the antecedent services to function correctly. The data in this class is derived by the provider from the registry key: System\CurrentControlSet\Control\GroupOrderList
Win32_LogicalDisk	Represents a data source that resolves to an actual local storage device on a Windows system.
Win32_LogonSession	Describes the logon session or sessions associated with a user logged on to Windows NT or Windows 2000.
Win32_CacheMemory	Represents internal and external cache memory on a computer system.
Win32_LogicalMemory Configuration	<p>Represents the layout and availability of memory on a Windows system. Beginning with Windows Vista, this class is no longer available in the operating system.</p> <p>Windows XP and Windows Server 2003: This class is no longer supported. Use the Win32_OperatingSystem class instead.</p> <p>Windows 2000: This class is available and supported.</p>

Class	Description
<code>Win32_PhysicalMemoryArray</code>	Represents details about the computer system's physical memory. This includes the number of memory devices, memory capacity available, and memory type—for example, system or video memory.
<code>Win32_NetworkClient</code>	Represents a network client on a Windows system. Any computer system on the network with a client relationship to the system is a descendant (or member) of this class (for example, a computer running Windows 2000 Workstation or Windows 98 that is part of a Windows 2000 domain).
<code>Win32_NetworkLoginProfile</code>	Represents the network login information of a specific user on a Windows system. This includes but is not limited to password status, access privileges, disk quotas, and login directory paths.
<code>Win32_NetworkProtocol</code>	Represents a protocol and its network characteristics on a Win32 computer system.
<code>Win32_NetworkConnection</code>	Represents an active network connection in a Windows environment.
<code>Win32_NetworkAdapter</code>	Represents a network adapter of a computer running on a Windows operating system.
<code>Win32_NetworkAdapterConfiguration</code>	Represents the attributes and behaviors of a network adapter. This class includes extra properties and methods that support the management of the TCP/IP and Internetworking Packet Exchange (IPX) protocols that are independent from the network adapter.
<code>Win32_NTDomain</code>	Represents a Windows NT domain.
<code>Win32_NTLogEvent</code>	Used to translate instances from the Windows NT event log. An application must have <code>SeSecurityPrivilege</code> to receive events from the security event log; otherwise, "Access Denied" is returned to the application.
<code>Win32_NTEventlogFile</code>	Represents a logical file or directory of Windows NT events. The file is also known as the event log.
<code>Win32_OnBoardDevice</code>	Represents common adapter devices built into the motherboard (system board).
<code>Win32_OperatingSystem</code>	Represents an operating system installed on a computer running on a Windows operating system. Any operating system that can be installed on a Windows system is a descendant or member of this class. <code>Win32_OperatingSystem</code> is a singleton class. To get the single instance, use @ for the key. Windows Server 2003, Windows XP, Windows 2000, and Windows NT 4.0: If a computer has multiple operating systems installed, this class returns only an instance for the currently active operating system.
<code>Win32_PageFileUsage</code>	Represents the file used for handling virtual memory file swapping on a Win32 system. Information contained within objects instantiated from this class specify the runtime state of the page file.

Class	Description
<code>Win32_PageFileSetting</code>	Represents the settings of a page file. Information contained within objects instantiated from this class specifies the page file parameters used when the file is created at system startup. The properties in this class can be modified and deferred until startup. These settings are different from the runtime state of a page file expressed through the associated class <code>Win32_PageFileUsage</code> .
<code>Win32_DiskPartition</code>	Represents the capabilities and management capacity of a partitioned area of a physical disk on a Windows system. Example: Disk #0, Partition #1.
<code>Win32_PortResource</code>	Represents an I/O port on a Windows computer system.
<code>Win32_PortConnector</code>	Represents physical connection ports, such as DB-25 pin male, Centronics, or PS/2.
<code>Win32_Printer</code>	Represents a device connected to a computer running on a Microsoft Windows operating system that can produce a printed image or text on paper or another medium.
<code>Win32_PrinterConfiguration</code>	Represents the configuration for a printer device. This includes capabilities such as resolution, color, fonts, and orientation.
<code>Win32_PrintJob</code>	Represents a print job generated by a Windows application. Any unit of work generated by the Print command of an application that is running on a computer running on a Windows operating system is a descendant or member of this class.
<code>Win32_Process</code>	Represents a process on an operating system.
<code>Win32_Product</code>	Represents products as they are installed by Windows Installer. A product generally correlates to one installation package. Note: for information about support or requirements for installation of a specific operating system, visit http://msdn.microsoft.com and search for "Operating System Availability of WMI Components."
<code>Win32_QuickFixEngineering</code>	Represents system-wide Quick Fix Engineering (QFE) or updates that have been applied to the current operating system.
<code>Win32_QuotaSetting</code>	Contains setting information for disk quotas on a volume.
<code>Win32_OSRecoveryConfiguration</code>	Represents the types of information that will be gathered from memory when the operating system fails. This includes boot failures and system crashes.
<code>Win32_Registry</code>	Represents the system registry on a Windows computer system.
<code>Win32_SCSIController</code>	Represents an SCSI controller on a Windows system.
<code>Win32_PerfRawData_PerfNet_Server</code>	Provides raw data from performance counters that monitor communications using the WINS Server service.
<code>Win32_Service</code>	Represents a service on a computer running on a Microsoft Windows operating system. A service application conforms to the interface rules of the Service Control Manager (SCM), and can be started by a user automatically at system start through the Services Control Panel utility, or by an application that uses the service functions included in the Windows API. Services can start when there are no users logged on to the computer.

Class	Description
Win32_Share	Represents a shared resource on a Windows system. This may be a disk drive, printer, interprocess communication, or other shareable device.
Win32_SoftwareElement	Represents a software element, part of a software feature (a distinct subset of a product, which may contain one or more elements). Each software element is defined in a Win32_SoftwareElement instance, and the association between a feature and its Win32_SoftwareFeature instance is defined in the Win32_SoftwareFeatureSoftware-Elements association class. Note: for information about support or requirements for installation on a specific operating system, visit http://msdn.microsoft.com and search for "Operating System Availability of WMI Components."
Win32_SoftwareFeature	Represents a distinct subset of a product that consists of one or more software elements. Each software element is defined in a Win32_SoftwareElement instance, and the association between a feature and its Win32_SoftwareFeature instance is defined in the Win32_SoftwareFeatureSoftware-Elements association class. Note: for information about support or requirements for installation on a specific operating system, visit http://msdn.microsoft.com and search for "Operating System Availability of WMI Components."
Win32_SoundDevice	Represents the properties of a sound device on a Windows computer system.
Win32_StartupCommand	Represents a command that runs automatically when a user logs on to the computer system.
Win32_SystemAccount	Represents a system account. The system account is used by the operating system and services that run under Windows NT. There are many services and processes within Windows NT that need the capability to log on internally, for example, during a Windows NT installation. The system account was designed for that purpose.
Win32_SystemDriver	Represents the system driver for a base service.
Win32_SystemEnclosure	Represents the properties that are associated with a physical system enclosure.
Win32_SystemSlot	Represents physical connection points, including ports, motherboard slots and peripherals, and proprietary connection points.
Win32_TapeDrive	Represents a tape drive on a Windows computer. Tape drives are primarily distinguished by the fact that they can be accessed only sequentially.
Win32_TemperatureProbe	Represents the properties of a temperature sensor (electronic thermometer).
Win32_TimeZone	Represents the time zone information for a Windows system, which includes changes required for the Daylight Savings Time transition.
Win32_UninterruptiblePowerSupply	Represents the capabilities and management capacity of an uninterruptible power supply (UPS). Beginning with Windows Vista, this class is obsolete and not available because the UPS service is no longer available. This service worked with serially attached UPS devices, not USB devices. Windows Server 2003 and Windows XP: This class is

Class	Description
	available but not usable because the UPS service fails. Windows Server 2003, Windows XP, Windows 2000, and Windows NT 4.0: This class is available and implemented.
<code>Win32_UserAccount</code>	Contains information about a user account on a computer running on a Windows operating system. Note: Because both the <code>Name</code> and <code>Domain</code> are key properties, enumerating <code>Win32_UserAccount</code> on a large network can affect performance negatively. Calling <code>GetObject</code> or querying for a specific instance has less impact.
<code>Win32_VoltageProbe</code>	Represents the properties of a voltage sensor (electronic voltmeter).
<code>Win32_VolumeQuotaSetting</code>	Relates disk quota settings with a specific disk volume. Windows 2000/NT: This class is not available.
<code>Win32_WMISetting</code>	Contains the operational parameters for the WMI service. This class can only have one instance, which always exists for each Windows system and cannot be deleted. Additional instances cannot be created.



Chapter 8. Selected COM Objects and Their Uses

As an extensibility and administration interface, many applications expose useful functionality through COM objects. While PowerShell handles many of these tasks directly, many COM objects still provide significant value.

Table 8-1 lists a selection of the COM objects most useful to system administrators.

Table 8-1. COM identifiers and descriptions

Identifier	Description
<code>Access.Application</code>	Allows for interaction and automation of Microsoft Access.
<code>Agent.Control</code>	Allows for the control of Microsoft Agent 3D-animated characters.
<code>AutoItX3.Control</code>	(Nondefault.) Provides access to Windows Automation via the <code>AutoIt</code> administration tool.
<code>CEnroll.CEnroll</code>	Provides access to certificate enrollment services.
<code>CertificateAuthority.Request</code>	Provides access to a request to a certificate authority.
<code>COMAdmin.COMAdminCatalog</code>	Provides access to and management of the Windows COM+ catalog.
<code>Excel.Application</code>	Allows for interaction and automation of Microsoft Excel.
<code>Excel.Sheet</code>	Allows for interaction with Microsoft Excel worksheets.
<code>HNetCfg.FwMgr</code>	Provides access to the management functionality of the Windows Firewall.
<code>HNetCfg.HNetShare</code>	Provides access to the management functionality of Windows Connection Sharing.
<code>HTMLFile</code>	Allows for interaction and authoring of a new Internet Explorer document.
<code>InfoPath.Application</code>	Allows for interaction and automation of Microsoft InfoPath.
<code>InternetExplorer.Application</code>	Allows for interaction and automation of Microsoft Internet Explorer.
<code>IXSSO.Query</code>	Allows for interaction with Microsoft Index Server.
<code>IXSSO.Util</code>	Provides access to utilities used along with the <code>IXSSO.Query</code> object.
<code>LegitCheckControl.LegitCheck</code>	Provides access to information about Windows Genuine Advantage status on the current computer.
<code>MakeCab.MakeCab</code>	Provides functionality to create and manage cabinet (<i>.cab</i>) files.
<code>MAPI.Session</code>	Provides access to a MAPI (Messaging Application Programming Interface) session, such as folders, messages, and the address book.
<code>Messenger.MessengerApp</code>	Allows for interaction and automation of Messenger.
<code>Microsoft.FeedsManager</code>	Allows for interaction with the Microsoft RSS feed platform.

Identifier	Description
<code>Microsoft.ISAdm</code>	Provides management of Microsoft Index Server.
<code>Microsoft.Update.AutoUpdate</code>	Provides management of the auto update schedule for Microsoft Update.
<code>Microsoft.Update.Installer</code>	Allows for installation of updates from Microsoft Update.
<code>Microsoft.Update.Searcher</code>	Provides search functionality for updates from Microsoft Update.
<code>Microsoft.Update.Session</code>	Provides access to local information about Microsoft Update history.
<code>Microsoft.Update.SystemInfo</code>	Provides access to information related to Microsoft Update for the current system.
<code>MMC20.Application</code>	Allows for interaction and automation of Microsoft Management Console (MMC).
<code>MSScriptControl.ScriptControl</code>	Allows for the evaluation and control of WSH scripts.
<code>Msxml2.XSLTemplate</code>	Allows for processing of XSL transforms.
<code>Outlook.Application</code>	Allows for interaction and automation of your email, calendar, contacts, tasks, and more through Microsoft Outlook.
<code>OutlookExpress.MessageList</code>	Allows for interaction and automation of your email through Microsoft Outlook Express.
<code>PowerPoint.Application</code>	Allows for interaction and automation of Microsoft PowerPoint.
<code>Publisher.Application</code>	Allows for interaction and automation of Microsoft Publisher.
<code>RDS.DataSpace</code>	Provides access to proxies of Remote DataSpace business objects.
<code>SAPI.SpVoice</code>	Provides access to the Microsoft Speech API.
<code>Scripting.FileSystemObject</code>	Provides access to the computer's filesystem. Most functionality is available more directly through PowerShell or through PowerShell's support for the .NET Framework.
<code>Scripting.Signer</code>	Provides management of digital signatures on WSH files.
<code>Scriptlet.TypeLib</code>	Allows the dynamic creation of scripting type library (<i>.tlb</i>) files.
<code>ScriptPW.Password</code>	Allows for the masked input of plain-text passwords. When possible, you should avoid this in preference of the <code>Read-Host</code> cmdlet with the <code>-AsSecureString</code> parameter.
<code>SharePoint.OpenDocuments</code>	Allows for interaction with Microsoft SharePoint Services.
<code>Shell.Application</code>	Provides access to aspects of the Windows Explorer Shell application, such as managing windows, files and folders, and the current session.
<code>Shell.LocalMachine</code>	Provides access to information about the current machine related to the Windows shell.
<code>Shell.User</code>	Provides access to aspects of the current user's Windows session and profile.
<code>SQLDMO.SQLServer</code>	Provides access to the management functionality of Microsoft SQL Server.

Identifier	Description
<code>Vim.Application</code>	(Nondefault.) Allows for interaction and automation of the VIM editor.
<code>WIA.CommonDialog</code>	Provides access to image capture through the Windows Image Acquisition facilities.
<code>WMPlayer.OCX</code>	Allows for interaction and automation of Windows Media Player.
<code>Word.Application</code>	Allows for interaction and automation of Microsoft Word.
<code>Word.Document</code>	Allows for interaction with Microsoft Word documents.
<code>WScript.Network</code>	Provides access to aspects of a networked Windows environment, such as printers and network drives, as well as computer and domain information.
<code>WScript.Shell</code>	Provides access to aspects of the Windows Shell, such as applications, shortcuts, environment variables, the registry, and operating environment.
<code>WSHController</code>	Allows the execution of WSH scripts on remote computers.





Chapter 9. .NET String Formatting

String Formatting Syntax

Standard Numeric Format Strings

Custom Numeric Format Strings

9.1. String Formatting Syntax

The format string supported by the format (`-f`) operator is a string that contains format items. Each format item takes the form of:

```
{index[,alignment][:formatString]}
```

`<index>` represents the zero-based index of the item in the object array following the format operator.

`<alignment>` is optional and represents the alignment of the item. A positive number aligns the item to the right of a field of the specified width. A negative number aligns the item to the left of a field of the specified width.

`<formatString>` is optional and formats the item using that type's specific format string syntax.





Chapter 9. .NET String Formatting

String Formatting Syntax

Standard Numeric Format Strings

Custom Numeric Format Strings

9.1. String Formatting Syntax

The format string supported by the format (`-f`) operator is a string that contains format items. Each format item takes the form of:

```
{index[,alignment][:formatString]}
```

`<index>` represents the zero-based index of the item in the object array following the format operator.

`<alignment>` is optional and represents the alignment of the item. A positive number aligns the item to the right of a field of the specified width. A negative number aligns the item to the left of a field of the specified width.

`<formatString>` is optional and formats the item using that type's specific format string syntax.





9.2. Standard Numeric Format Strings

Table 9-1 lists the standard numeric format strings. All format specifiers may be followed by a number between 0 and 99 to control the precision of the formatting.

Table 9-1. Standard numeric format strings

Format specifier (Name)	Description	Example
C or c (Currency)	A currency amount.	<code>PS > "{0:C}" -f 1.23</code> \$1.23
D or d (Decimal)	A decimal amount (for integral types). The precision specifier controls the minimum number of digits in the result.	<code>PS > "{0:D4}" -f 2</code> 0002
E or e (Scientific)	Scientific (exponential) notation. The precision specifier controls the number of digits past the decimal point.	<code>PS > "{0:E3}" -f [Math]::Pi</code> 3.142E+000
F or f (Fixed-point)	Fixed point notation. The precision specifier controls the number of digits past the decimal point.	<code>PS > "{0:F3}" -f [Math]::Pi</code> 3.142
G or g (General)	The most compact representation (between fixed-point and scientific) of the number. The precision specifier controls the number of significant digits.	<code>PS > "{0:G3}" -f [Math]::Pi</code> 3.14 <code>PS > "{0:G3}" -f 1mb</code> 1.05E+06
N or n (Number)	The human readable form of the number, which includes separators between number groups. The precision specifier controls the number of digits past the decimal point.	<code>PS > "{0:N4}" -f 1mb</code> 1,048,576.0000
P or p (Percent)	The number (generally between 0 and 1) represented as a percentage. The precision specifier controls the number of digits past the decimal point.	<code>PS > "{0:P4}" -f 0.67</code> 67.0000 %
R or r (Round-trip)	The single or double number formatted with a precision that guarantees the string (when parsed) will result in the original number again.	<code>PS > "{0:R}" -f (1mb/2.0)</code> 524288 <code>PS > "{0:R}" -f (1mb/9.0)</code> 116508.444444444444
X or x (Hexadecimal)	The number converted to a string of hexadecimal digits. The case of the specifier controls the case of the resulting hexadecimal digits. The precision specifier controls the minimum number of digits in the resulting string.	<code>PS > "{0:X4}" -f 1324</code> 052C





9.3. Custom Numeric Format Strings

You may use custom numeric format strings, listed in [Table 9-2](#), to format numbers in ways not supported by the standard format strings.

Table 9-2. Custom numeric format strings

Format specifier (Name)	Description	Example
0 (Zero placeholder)	Specifies the precision and width of a number string. Zeroes not matched by digits in the original number are output as zeroes.	PS > "{0:00.0}" -f 4.12341234 04.1
# (Digit placeholder)	Specifies the precision and width of a number string. # symbols not matched by digits in the input number are not output.	PS > "{0:##.}" -f 4.12341234 4.1
. (Decimal point)	Determines the location of the decimal separator.	PS > "{0:##.}" -f 4.12341234 4.1
, (Thousands separator)	When placed between a zero or digit placeholder before the decimal point in a formatting string, adds the separator character between number groups.	PS > "{0:#,#.}" -f 1234.121234 1,234.1
, (Number scaling)	When placed before the literal (or implicit) decimal point in a formatting string, divides the input by 1,000. You may apply this format specifier more than once.	PS > "{0:##,.,.000}" -f 1048576 1.049
% (Percentage placeholder)	Multiplies the input by 100 and inserts the percent sign where shown in the format specifier.	PS > "{0:%##.000}" -f .68 %68.000
E0 E+0 E-0 e0 e+0 e-0 (Scientific notation)	Displays the input in scientific notation. The number of zeroes that follow the E define the minimum length of the exponent field.	PS > "{0:##.##E000}" -f 2.71828 27.2E-001
'text' "text" (Literal string)	Inserts the provided text literally into the output without affecting formatting.	PS > "{0:#.00'##'}" -f 2.71828 2.72##

Format specifier (Name)	Description	Example
; (Section separator)	Allows for conditional formatting. If your format specifier contains no section separators, then the formatting statement applies to all input. If your format specifier contains one separator (creating two sections), then the first section applies to positive numbers and zero. The second section applies to negative numbers. If your format specifier contains two separators (creating three sections), then the sections apply to positive numbers, negative numbers, and zero.	<pre>PS > "{0:POS;NEG;ZERO}"-f -14 NEG</pre>
Other (Other character)	Inserts the provided text literally into the output without affecting formatting.	<pre>PS > "{0:\$## Please}"-f 14 \$14 Please</pre>





Chapter 10. .NET DateTime Formatting

DateTime format strings convert a `DateTime` object to one of several standard formats, as listed in Table 10-1.

Table 10-1. Standard DateTime format strings

Format specifier (Name)	Description	Example
d (Short date)	The culture's short date format.	<code>PS >"{0:d}" -f [DateTime] "01/23/4567"</code> 1/23/4567
D (Long date)	The culture's long date format.	<code>PS >"{0:D}" -f [DateTime] "01/23/4567"</code> Friday, January 23, 4567
f (Full date/short time)	Combines the long date and short time format patterns.	<code>PS >"{0:f}" -f [DateTime] "01/23/4567"</code> Friday, January 23, 4567 12:00 AM
F (Full date/long time)	Combines the long date and long time format patterns.	<code>PS >"{0:F}" -f [DateTime] "01/23/4567"</code> Friday, January 23, 4567 12:00:00 AM
g (General date/short time)	Combines the short date and short time format patterns.	<code>PS >"{0:g}" -f [DateTime] "01/23/4567"</code> 1/23/4567 12:00 AM
G (General date/long time)	Combines the short date and long time format patterns.	<code>PS >"{0:G}" -f [DateTime] "01/23/4567"</code> 1/23/4567 12:00:00 AM
M or m (Month day)	The culture's <code>MonthDay</code> format.	<code>PS >"{0:M}" -f [DateTime] "01/23/4567"</code> January 23
o (Round-trip date/time)	The date formatted with a pattern that guarantees the string (when parsed) will result in the original <code>DateTime</code> again.	<code>PS >"{0:o}" -f [DateTime] "01/23/4567"</code> 4567-01-23T00:00:00.0000000
R or r (RFC1123)	The standard RFC1123 format pattern.	<code>PS >"{0:R}" -f [DateTime] "01/23/4567"</code> Fri, 23 Jan 4567 00:00:00 GMT
s (Sortable)	Sortable format pattern. Conforms to ISO 8601 and provides output suitable for sorting.	<code>PS >"{0:s}" -f [DateTime] "01/23/4567"</code> 4567-01-23T00:00:00
t (Short time)	The culture's short time format.	<code>PS >"{0:t}" -f [DateTime] "01/23/4567"</code> 12:00 AM

Format specifier (Name)	Description	Example
T (Long time)	The culture's long time format.	PS >"{0:T}" -f [DateTime] "01/23/4567" 12:00:00 AM
u (Universal sortable)	The culture's universal sortable <code>DateTime</code> format applied to the UTC equivalent of the input.	PS >"{0:u}" -f [DateTime] "01/23/4567" 4567-01-23 00:00:00Z
U (Universal)	The culture's <code>FullDate-Time</code> format applied to the UTC equivalent of the input.	PS >"{0:U}" -f [DateTime] "01/23/4567" Friday, January 23, 4567 8:00:00 AM
Y or y (Year month)	The culture's <code>YearMonth</code> format.	PS >"{0:Y}" -f [DateTime] "01/23/4567" January, 4567

10.1. Custom DateTime Format Strings

You may use custom `DateTime` format strings, listed in [Table 10-2](#), to format dates in ways not supported by the standard format strings. Note: Single-character format specifiers are interpreted as a standard `DateTime` formatting string unless used with other formatting specifiers.

Table 10-2. Custom `DateTime` format strings

Format specifier	Description	Example
d	Day of the month as a number between 1 and 31. Represents single-digit days without a leading zero.	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday
dd	Day of the month as a number between 1 and 31. Represents single-digit days with a leading zero.	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday
ddd	Abbreviated name of the day of the week.	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday
dddd	Full name of the day of the week.	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday

Format specifier	Description	Example
<code>f</code>	Most significant digit of the seconds fraction (milliseconds).	<pre>PS >"{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000</pre>
<code>ff</code>	Two most significant digits of the seconds fraction (milliseconds).	<pre>PS >"{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000</pre>
<code>fff</code>	Three most significant digits of the seconds fraction (milliseconds).	<pre>PS >"{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000</pre>
<code>ffff</code>	Four most significant digits of the seconds fraction (milliseconds).	<pre>PS >"{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000</pre>
<code>fffff</code>	Five most significant digits of the seconds fraction (milliseconds).	<pre>PS >"{0:fffff fffffff ffffffff}" -f [DateTime] "01/02/4567" 00000 000000 0000000</pre>
<code>ffffff</code>	Six most significant digits of the seconds fraction (milliseconds).	<pre>PS >"{0:ffffff fffffff ffffffff}" -f [DateTime] "01/02/4567" 00000 000000 0000000</pre>
<code>fffffff</code>	Seven most significant digits of the seconds fraction (milliseconds).	<pre>PS >"{0:fffffff fffffff ffffffff}" -f [DateTime] "01/02/4567" 00000 000000 0000000</pre>
<code>F</code>	Most significant digit of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS >"{0:F FF FFF FFFF}" -f [DateTime]::Now 6 66 669 6696 PS >"{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567" </pre>
<code>FF</code>	Two most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS >"{0:F FF FFF FFFF}" -f [DateTime]::Now 6 66 669 6696 PS >"{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567" </pre>
<code>FFF</code>	Three most significant digits of the seconds	

Format specifier	Description	Example
FFF	Three most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:F FF FFF FFFF}" -f [DateTime]::Now 6 66 669 6696 PS > "{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567" </pre>
FFFF	Four most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:F FF FFF FFFF}" -f [DateTime]::Now 6 66 669 6696 PS > "{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567" </pre>
FFFFF	Five most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:FFFFFF FFFFFFF FFFFFFF}" -f [DateTime]::Now 1071 107106 1071068 PS > "{0: FFFFFF FFFFFFF FFFFFFF }" -f [DateTime] "01/02/4567" </pre>
FFFFFF	Six most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:FFFFFF FFFFFFF FFFFFFF}" -f [DateTime]::Now 1071 107106 1071068 PS > "{0: FFFFFF FFFFFFF FFFFFFF }" -f [DateTime] "01/02/4567" </pre>
FFFFFFF	Seven most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:FFFFFF FFFFFFF FFFFFFF}" -f [DateTime]::Now 1071 107106 1071068 PS > "{0: FFFFFF FFFFFFF FFFFFFF }" -f [DateTime] "01/02/4567" </pre>
%g or gg	Era (i.e., A.D.).	<pre>PS > "{0:gg}" -f [DateTime] "01/02/4567" A.D.</pre>
%h	Hours, as a number between 1 and 12. Single digits do not include a leading zero.	<pre>PS > "{0:%h}" -f [DateTime] "01/02/4567 4:00pm" 4</pre>

Format specifier	Description	Example
<code>hh</code>	Hours, as a number between 01 and 12. Single digits include a leading zero. Note: This is interpreted as a standard <code>DateTime</code> formatting string unless used with other formatting specifiers.	PS > "{0:hh}" -f [DateTime] "01/02/4567 4:00pm" 04
<code>%H</code>	Hours, as a number between 0 and 23. Single digits do not include a leading zero.	PS > "{0:%H}" -f [DateTime] "01/02/4567 4:00pm" 16
<code>HH</code>	Hours, as a number between 00 and 23. Single digits include a leading zero.	PS > "{0:HH}" -f [DateTime] "01/02/4567 4:00am" 04
<code>K</code>	<code>DateTime.Kind</code> specifier that corresponds to the kind (i.e., Local, Utc, or Unspecified) of input date.	PS > "{0: K}" -f [DateTime]::Now.ToUniversalTime() Z
<code>m</code>	Minute, as a number between 0 and 59. Single digits do not include a leading zero.	PS > "{0: m}" -f [DateTime]::Now 7
<code>mm</code>	Minute, as a number between 00 and 59. Single digits include a leading zero.	PS > "{0:mm}" -f [DateTime]::Now 08
<code>M</code>	Month, as a number between 1 and 12. Single digits do not include a leading zero.	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
<code>MM</code>	Month, as a number between 01 and 12. Single digits include a leading zero.	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
<code>MMM</code>	Abbreviated month name.	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
<code>MMMM</code>	Full month name.	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
<code>s</code>	Seconds, as a number between 0 and 59. Single digits do not include a leading zero.	PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM

Format specifier	Description	Example
<code>ss</code>	Seconds, as a number between 00 and 59. Single digits include a leading zero.	<pre>PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM</pre>
<code>t</code>	First character of the A.M./P.M. designator.	<pre>PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM</pre>
<code>tt</code>	A.M./P.M designator.	<pre>PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM</pre>
<code>y</code>	Year, in (at most) one digit.	<pre>PS > "{0:y yy yyy YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567</pre>
<code>yy</code>	Year, in (at most) two digits.	<pre>PS > "{0:y yy yyy YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567</pre>
<code>yyy</code>	Year, in (at most) three digits.	<pre>PS > "{0:y yy yyy YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567</pre>
<code>yyyy</code>	Year, in (at most) four digits.	<pre>PS > "{0:y yy yyy YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567</pre>
<code>yyyyy</code>	Year, in (at most) five digits.	<pre>PS > "{0:y yy yyy YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567</pre>
<code>z</code>	Signed time zone offset from GMT. Does not include a leading zero.	<pre>PS > "{0:z zz zzz}" -f [DateTime]::Now -7 -07 -07:00</pre>
<code>zz</code>	Signed time zone offset from GMT. Includes a leading zero.	<pre>PS > "{0:z zz zzz}" -f [DateTime]::Now -7 -07 -07:00</pre>
<code>zzz</code>	Signed time zone offset from GMT, measured in hours and minutes.	<pre>PS > "{0:z zz zzz}" -f [DateTime]::Now -7 -07 -07:00</pre>

Format specifier	Description	Example
:	Time separator.	<pre>PS > "{0:y/m/d h:m:s}" -f [DateTime] "01/02/4567 4:00pm" 67/0/2 4:0:0</pre>
/	Date separator.	<pre>PS > "{0:y/m/d h:m:s}" -f [DateTime] "01/02/4567 4:00pm" 67/0/2 4:0:0</pre>
"text" 'text'	Inserts the provided text literally into the output without affecting formatting.	<pre>PS > "{0:'Day: 'dddd}" -f [DateTime]::Now Day: Monday</pre>
%c	Syntax allowing for single-character custom formatting specifiers. The % sign is not added to the output.	<pre>PS > "{0:%h}" -f [DateTime] "01/02/4567 4:00pm" 4</pre>
Other	Inserts the provided text literally into the output without affecting formatting.	<pre>PS > "{0:dddd!}" -f [DateTime]::Now Monday!</pre>





Chapter 10. .NET DateTime Formatting

DateTime format strings convert a `DateTime` object to one of several standard formats, as listed in Table 10-1.

Table 10-1. Standard DateTime format strings

Format specifier (Name)	Description	Example
<code>d</code> (Short date)	The culture's short date format.	<code>PS >"{0:d}" -f [DateTime] "01/23/4567"</code> 1/23/4567
<code>D</code> (Long date)	The culture's long date format.	<code>PS >"{0:D}" -f [DateTime] "01/23/4567"</code> Friday, January 23, 4567
<code>f</code> (Full date/ short time)	Combines the long date and short time format patterns.	<code>PS >"{0:f}" -f [DateTime] "01/23/4567"</code> Friday, January 23, 4567 12:00 AM
<code>F</code> (Full date/ long time)	Combines the long date and long time format patterns.	<code>PS >"{0:F}" -f [DateTime] "01/23/4567"</code> Friday, January 23, 4567 12:00:00 AM
<code>g</code> (General date/ short time)	Combines the short date and short time format patterns.	<code>PS >"{0:g}" -f [DateTime] "01/23/4567"</code> 1/23/4567 12:00 AM
<code>G</code> (General date/ long time)	Combines the short date and long time format patterns.	<code>PS >"{0:G}" -f [DateTime] "01/23/4567"</code> 1/23/4567 12:00:00 AM
<code>M</code> or <code>m</code> (Month day)	The culture's <code>MonthDay</code> format.	<code>PS >"{0:M}" -f [DateTime] "01/23/4567"</code> January 23
<code>o</code> (Round-trip date/time)	The date formatted with a pattern that guarantees the string (when parsed) will result in the original <code>DateTime</code> again.	<code>PS >"{0:o}" -f [DateTime] "01/23/4567"</code> 4567-01-23T00:00:00.0000000
<code>R</code> or <code>r</code> (RFC1123)	The standard RFC1123 format pattern.	<code>PS >"{0:R}" -f [DateTime] "01/23/4567"</code> Fri, 23 Jan 4567 00:00:00 GMT
<code>s</code> (Sortable)	Sortable format pattern. Conforms to ISO 8601 and provides output suitable for sorting.	<code>PS >"{0:s}" -f [DateTime] "01/23/4567"</code> 4567-01-23T00:00:00
<code>t</code> (Short time)	The culture's short time format.	<code>PS >"{0:t}" -f [DateTime] "01/23/4567"</code> 12:00 AM

Format specifier (Name)	Description	Example
T (Long time)	The culture's long time format.	PS >"{0:T}" -f [DateTime] "01/23/4567" 12:00:00 AM
u (Universal sortable)	The culture's universal sortable <code>DateTime</code> format applied to the UTC equivalent of the input.	PS >"{0:u}" -f [DateTime] "01/23/4567" 4567-01-23 00:00:00Z
U (Universal)	The culture's <code>FullDate-Time</code> format applied to the UTC equivalent of the input.	PS >"{0:U}" -f [DateTime] "01/23/4567" Friday, January 23, 4567 8:00:00 AM
Y or y (Year month)	The culture's <code>YearMonth</code> format.	PS >"{0:Y}" -f [DateTime] "01/23/4567" January, 4567

10.1. Custom DateTime Format Strings

You may use custom `DateTime` format strings, listed in [Table 10-2](#), to format dates in ways not supported by the standard format strings. Note: Single-character format specifiers are interpreted as a standard `DateTime` formatting string unless used with other formatting specifiers.

Table 10-2. Custom `DateTime` format strings

Format specifier	Description	Example
d	Day of the month as a number between 1 and 31. Represents single-digit days without a leading zero.	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday
dd	Day of the month as a number between 1 and 31. Represents single-digit days with a leading zero.	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday
ddd	Abbreviated name of the day of the week.	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday
dddd	Full name of the day of the week.	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday

Format specifier	Description	Example
<code>f</code>	Most significant digit of the seconds fraction (milliseconds).	<pre>PS > "{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000</pre>
<code>ff</code>	Two most significant digits of the seconds fraction (milliseconds).	<pre>PS > "{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000</pre>
<code>fff</code>	Three most significant digits of the seconds fraction (milliseconds).	<pre>PS > "{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000</pre>
<code>ffff</code>	Four most significant digits of the seconds fraction (milliseconds).	<pre>PS > "{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000</pre>
<code>fffff</code>	Five most significant digits of the seconds fraction (milliseconds).	<pre>PS > "{0:fffff fffffff ffffffff}" -f [DateTime] "01/02/4567" 00000 000000 0000000</pre>
<code>ffffff</code>	Six most significant digits of the seconds fraction (milliseconds).	<pre>PS > "{0:ffffff fffffff ffffffff}" -f [DateTime] "01/02/4567" 00000 000000 0000000</pre>
<code>fffffff</code>	Seven most significant digits of the seconds fraction (milliseconds).	<pre>PS > "{0:fffffff fffffff ffffffff}" -f [DateTime] "01/02/4567" 00000 000000 0000000</pre>
<code>F</code>	Most significant digit of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:F FF FFF FFFF}" -f [DateTime]::Now 6 66 669 6696 PS > "{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567" </pre>
<code>FF</code>	Two most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:F FF FFF FFFF}" -f [DateTime]::Now 6 66 669 6696 PS > "{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567" </pre>
<code>FFF</code>	Three most significant digits of the seconds	

Format specifier	Description	Example
FFF	Three most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:F FF FFF FFFF}" -f [DateTime]::Now 6 66 669 6696 PS > "{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567" </pre>
FFFF	Four most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:F FF FFF FFFF}" -f [DateTime]::Now 6 66 669 6696 PS > "{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567" </pre>
FFFFF	Five most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:FFFFFF FFFFFFF FFFFFFF}" -f [DateTime]::Now 1071 107106 1071068 PS > "{0: FFFFFF FFFFFFF FFFFFFF }" -f [DateTime] "01/02/4567" </pre>
FFFFFF	Six most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:FFFFFF FFFFFFF FFFFFFF}" -f [DateTime]::Now 1071 107106 1071068 PS > "{0: FFFFFF FFFFFFF FFFFFFF }" -f [DateTime] "01/02/4567" </pre>
FFFFFFF	Seven most significant digits of the seconds fraction (milliseconds). Displays nothing if the number is zero.	<pre>PS > "{0:FFFFFF FFFFFFF FFFFFFF}" -f [DateTime]::Now 1071 107106 1071068 PS > "{0: FFFFFF FFFFFFF FFFFFFF }" -f [DateTime] "01/02/4567" </pre>
%g or gg	Era (i.e., A.D.).	<pre>PS > "{0:gg}" -f [DateTime] "01/02/4567" A.D.</pre>
%h	Hours, as a number between 1 and 12. Single digits do not include a leading zero.	<pre>PS > "{0:%h}" -f [DateTime] "01/02/4567 4:00pm" 4</pre>

Format specifier	Description	Example
<code>hh</code>	Hours, as a number between 01 and 12. Single digits include a leading zero. Note: This is interpreted as a standard <code>DateTime</code> formatting string unless used with other formatting specifiers.	PS > "{0:hh}" -f [DateTime] "01/02/4567 4:00pm" 04
<code>%H</code>	Hours, as a number between 0 and 23. Single digits do not include a leading zero.	PS > "{0:%H}" -f [DateTime] "01/02/4567 4:00pm" 16
<code>HH</code>	Hours, as a number between 00 and 23. Single digits include a leading zero.	PS > "{0:HH}" -f [DateTime] "01/02/4567 4:00am" 04
<code>K</code>	<code>DateTime.Kind</code> specifier that corresponds to the kind (i.e., Local, Utc, or Unspecified) of input date.	PS > "{0: K}" -f [DateTime]::Now.ToUniversalTime() Z
<code>m</code>	Minute, as a number between 0 and 59. Single digits do not include a leading zero.	PS > "{0: m}" -f [DateTime]::Now 7
<code>mm</code>	Minute, as a number between 00 and 59. Single digits include a leading zero.	PS > "{0:mm}" -f [DateTime]::Now 08
<code>M</code>	Month, as a number between 1 and 12. Single digits do not include a leading zero.	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
<code>MM</code>	Month, as a number between 01 and 12. Single digits include a leading zero.	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
<code>MMM</code>	Abbreviated month name.	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
<code>MMMM</code>	Full month name.	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
<code>s</code>	Seconds, as a number between 0 and 59. Single digits do not include a leading zero.	PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM

Format specifier	Description	Example
<code>ss</code>	Seconds, as a number between 00 and 59. Single digits include a leading zero.	<pre>PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM</pre>
<code>t</code>	First character of the A.M./P.M. designator.	<pre>PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM</pre>
<code>tt</code>	A.M./P.M designator.	<pre>PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM</pre>
<code>y</code>	Year, in (at most) one digit.	<pre>PS > "{0:y yy yyy YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567</pre>
<code>yy</code>	Year, in (at most) two digits.	<pre>PS > "{0:y yy yyy YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567</pre>
<code>yyy</code>	Year, in (at most) three digits.	<pre>PS > "{0:y yy yyy YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567</pre>
<code>yyyy</code>	Year, in (at most) four digits.	<pre>PS > "{0:y yy yyy YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567</pre>
<code>yyyyy</code>	Year, in (at most) five digits.	<pre>PS > "{0:y yy yyy YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567</pre>
<code>z</code>	Signed time zone offset from GMT. Does not include a leading zero.	<pre>PS > "{0:z zz zzz}" -f [DateTime]::Now -7 -07 -07:00</pre>
<code>zz</code>	Signed time zone offset from GMT. Includes a leading zero.	<pre>PS > "{0:z zz zzz}" -f [DateTime]::Now -7 -07 -07:00</pre>
<code>zzz</code>	Signed time zone offset from GMT, measured in hours and minutes.	<pre>PS > "{0:z zz zzz}" -f [DateTime]::Now -7 -07 -07:00</pre>

Format specifier	Description	Example
:	Time separator.	<pre>PS > "{0:y/m/d h:m:s}" -f [DateTime] "01/02/4567 4:00pm" 67/0/2 4:0:0</pre>
/	Date separator.	<pre>PS > "{0:y/m/d h:m:s}" -f [DateTime] "01/02/4567 4:00pm" 67/0/2 4:0:0</pre>
"text" 'text'	Inserts the provided text literally into the output without affecting formatting.	<pre>PS > "{0:'Day: 'dddd}" -f [DateTime]::Now Day: Monday</pre>
%c	Syntax allowing for single-character custom formatting specifiers. The % sign is not added to the output.	<pre>PS > "{0:%h}" -f [DateTime] "01/02/4567 4:00pm" 4</pre>
Other	Inserts the provided text literally into the output without affecting formatting.	<pre>PS > "{0:dddd!}" -f [DateTime]::Now Monday!</pre>





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- ! (exclamation point)
- # (comment character) 2nd
- % (percent sign)
 - +?
 - addition operator
 - in array ranges
- %g format specifier
- %H format specifier
- %h format specifier
- & (ampersand)
- ' (backtick)
- ' (single quote)
- (...) (parentheses)
 - @(...) array definition
- . (dot)
 - in regular expressions
 - method and property access
- (double quote)
 - %=
 - =
 - =
 - =
- (hyphen) 2nd
 - subtraction operator
- * (asterisk)
 - *=
 - *?
 - in regular expressions
 - multiplication operator
- 0x prefix
- : (colon)
- =+ (plus sign)
 - addition assignment
 - addition operator (+)
 - in array ranges
- > (right angle bracket)
- ? (question mark)
 - ?!
 - ?#
 - ?:
 - ?<!
 - ?<=
 - ?=
 - ?>
 - ??
 - in regular expressions 2nd
- @ (at sign)
 - @(...) list evaluation control
- [...] (square brackets)
 - array access using
- \ (backslash)
- \A
- \B

`\b`
`\D`
`\d`
`\G`
`\k`
`\P`
`\p`
`\S`
`\s`
`\W`
`\w`
`\Z`
`\z`
`^` (caret) 2nd 3rd
`{...}` (curly braces)
in regular expressions 2nd
in variable definition
statement block





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- Access.Application object
- Active Directory
- Active Directory Service Interfaces (ADSI) support
- Add verb
- Add-Member cmdlet 2nd 3rd 4th
- administrative tasks 2nd
- ADSI (Active Directory Service Interfaces) support
- Adsi type shortcut
- Agent.Control object
- alarm character
- aliases for cmdlets
- AliasProperty type
- Alt + F7 hotkey
- Alt + space 2nd 3rd 4th 5th
- alternation constructs
- and (logical AND operator)
- AppDomain class
- arbitrary variables
- arithmetic operators 2nd
- Array class
- ArrayList class
- arrays
 - accessing elements of
 - associative (hashtables)
 - data type of
 - numbering of elements in
 - ranges in 2nd
 - slicing
- as (type conversion operator)
- Assembly class 2nd
- assignment operators
- associative arrays (hashtables)
- asterisk (*)
 - *?
- atomic zero-width assertions
- auto-complete for cmdlets
- AutoHotkey program
- AutoItX3.Control object
- automatic variables 2nd





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- backreference constructs
- backspace character
- Backup verb
- band (binary AND operator)
- begin keyword in scripts
- begin statement
- binary exclusive OR operator (-bxor)
- binary NOT operator (-bnot)
- binary numbers
- binary operators 2nd 3rd
- binary OR operator (-bor)
- BinaryReader class
- BinaryWriter class
- Bitmap class
- bnot (binary NOT operator)
- Boolean values
- Boolean variables
- bor (binary OR operator)
- break keyword
- break statement
- BufferedStream class
- bxor (binary exclusive OR operator)





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- C format specifier
- c option
- c prefix
- calendar calculations
- cancel
- carriage return
- case sensitivity
 - for comparison operators
 - in switch statement
- casesensitive option
- cd command
- CEnroll.CEnroll object
- certificate store
- CertificateAuthority.Request object
- character classes
- Checkpoint verb
- CIM_DataFile class
- classes
 - assembly for
 - extending 2nd
 - information about
 - instances of
 - WMI classes 2nd 3rd 4th 5th
- Clear verb
- cmdlets 2nd
 - in scripts
 - positional parameters for
- CodeMethod type
- CodeProperty type
- collections
- COM objects 2nd 3rd 4th 5th
- COMAdmin.COMAdminCatalog object
- command history
- command line
- commands
 - formatting output for 2nd 3rd 4th
 - information about
 - objects generated by
 - output of 2nd
 - Unix
- comments
 - in scripts
- communication
 - ComObject parameter
- Compare verb
- comparison operators 2nd 3rd
- conditional statements 2nd 3rd 4th 5th 6th
- configuration files
 - for extending types
 - for output formatting
- Confirm parameter
- Connect verb
- Console class
- contains operator
- Continue ErrorAction preference

- continue keyword
- continue statement
- Continue verbose preference
- Control + break hotkey
- Control + c hotkey
- Control + end hotkey
- Control + home hotkey
- Control + left arrow hotkey
- Control + right arrow hotkey
- Convert class
- Convert verb
- ConvertFrom verb
- ConvertTo verb
- Copy verb
- CSharpCodeProvider class
- curly braces ({...})
 - statement block
- customization
 - hotkeys 2nd
 - of console 2nd 3rd 4th
 - QuickEdit mode
 - window size





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- D format specifier 2nd
- d format specifier
- data
- data type conversions
 - operator for
 - to integer
- data types
 - of array elements
 - XML
- database
- DataSet class
- DataTable class
- date calculations
- DateTime class
- DateTime format strings 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
- dd format specifier
- ddd format specifier
- dddd format specifier
- Debug class
- Debug verb
- debugging
- decimal numbers
- DeflateStream class
- dir command
- Directory class
- DirectoryEntry class 2nd
- DirectoryInfo class
- DirectorySearcher class
- Disable verb
- Disconnect verb
- Dismount verb
- division assignment operator (/=)
- division operator (/)
- Dns class
- do until statement
- do while statement
- DOS commands
- double quote ("")
- double quotes ("...")
- Down arrow hotkey





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- E format specifier
- e option
- else statement
- elseif statement
- Enable verb
- end keyword in scripts
- end statement
- Enum class
- Environment class
- equality operator (-ed)
- error codes returned from scripts
- error output stream
- ErrorAction parameter for cmdlets
- errors
 - nonterminating
- escape sequences
 - in strings
- evaluation controls
- EventLog class
- exact option
- Excel.Application object
- Excel.Sheet object
- exit statement
- expanding strings
- explicit capture
- Export verb
- expression subparse control (
- expressions
 - in expanding strings





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- f format operator 2nd 3rd 4th
- F format specifier 2nd
- f format specifier
- F1 hotkey
- F2 hotkey
- F3 hotkey
- F4 hotkey
- F5 hotkey
- F8 hotkey
- F9 hotkey
- FF format specifier
- ff format specifier
- FFF format specifier
- fff format specifier
- FFFF format specifier
- ffff format specifier
- FFFFF format specifier
- fffff format specifier
- FFFFFF format specifier
- ffffff format specifier
- File class
- FileInfo class
- files
 - command output in
 - getting and setting content as variables
- filesystem
- FileSystemSecurity class
- FileSystemWatcher class
- FlowLayoutPanel class
- for statement
- foreach statement
- Form class
- form feed
- format operator (-f)
- Format-List cmdlet
- Format-Table cmdlet
- Format-Wide cmdlet
- .Format.Ps1Xml file extension
- formatting command output 2nd 3rd 4th
- formatting files
- FtpWebRequest class
- functions 2nd
 - location of





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- G format specifier 2nd
- g format specifier
- gb (gigabyte) constant 2nd
- ge (greater than or equal operator)
- Get verb
- Get-Command cmdlet
- Get-Help cmdlet
- Get-History cmdlet
- Get-Item Variable cmdlet
- Get-Member cmdlet 2nd
- Get-Process cmdlet
- Get-TraceSource cmdlet
- Get-Variable cmdlet
- gg format specifier
- gigabyte (gb) constant 2nd
- gps alias
- greater than operator (-gt)
- grouping constructs
- gt (greater than operator)
- Guid class
- GZipStream class





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- Hashtable class
- hashtable definition (@{...})
- Hashtable type shortcut
- hashtables
- here strings 2nd
- hexadecimal base
- hexadecimal numbers
- HH format specifier
- Hide verb
- history buffer
- history of commands
- HNetCfg.FwMgr object
- HNetCfg.HNetShare object
- hotkeys 2nd 3rd 4th
- HTMLFile object
- HttpUtility class
- HttpWebRequest class





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- i prefix
- Image class
- Import verb
- InfoPath.Application object
- Initialize verb
- Inquire verbose preference
- Install verb
- instance methods
- instance properties
- int data type
- interactive shell 2nd 3rd 4th 5th
- InternetExplorer.Application object
- Invoke verb
- invoke/call operator (&)
- ipconfig tool
- is (type operator)
- isnot (negated type operator)
- IXSSO.Query object
- IXSSO.Util object





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

jagged arrays

Join verb





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

K format specifier
kb (kilobyte) constant
kilobyte (kb) constant





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- le (less than or equal operator)
- LegitCheckControl.LegitCheck object
- less than operator (-lt)
- like operator
- Limit verb
- list evaluation control (@(...))
- literal strings
- LoadWithPartialName method
- Lock verb
- logic statements
- logical exclusive OR operator (-xor)
- logical NOT operator (-not or !)
- logical operators
- logical OR operator (-or)
- looping statements 2nd 3rd 4th 5th 6th
 - halting execution of
 - skipping execution of current statement block
- lt (less than operator)





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- M format specifier 2nd
- m format specifier
- MailAddress class
- MailMessage class
- MakeCab.MakeCab object
- ManagementClass class 2nd
- ManagementDateTimeConverter class
- ManagementEventWatcher class
- ManagementObject class 2nd
- ManagementObjectSearcher class 2nd
- MAPI.Session object
- Marshal class
- match operator
- Math class
- mb (megabyte) constant 2nd
- Measure verb
- megabyte (mb) constant 2nd
- MemoryStream class
- Merge verb
- Messenger.MessengerApp object
- methods
 - adding to types 2nd 3rd 4th
 - instance
 - listing for types or classes
 - static
- Microsoft.FeedsManager object
- Microsoft.ISAdm object
- Microsoft.Update.AutoUpdate object
- Microsoft.Update.Installer object
- Microsoft.Update.Searcher object
- Microsoft.Update.Session object
- Microsoft.Update.SystemInfo object
- MM format specifier
- MMC20.Application object
- MMM format specifier
- MMMM format specifier
- modulus assignment operator (%=)
- modulus operator (%)
- Move verb
- MSScriptControl.ScriptControl object
- Msxml2.XSLTemplate object
- multidimensional arrays
- multiplication assignment operator (*=)
- multiplication operator (*)





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- N format specifier
- ne (negated equality operator)
- negated contains operator (-notcontains)
- negated equality operator (-ne)
- negated like operator (-notlike)
- negated match operator (-notmatch)
- negated type operator (-isnot)
- .NET Framework
 - documentation for
 - methods
 - properties
 - types and classes
 - creating instances of
- .NET Framework classes
 - Active Directory
 - collections
 - database
 - image manipulation
 - PowerShell object
 - registry
 - security
 - user interface
 - WMI
 - XML
- NetworkCredential class
- networking
- New verb
- New-Item Variable cmdlet
- New-Object cmdlet 2nd
- New-Variable cmdlet
- newline
- nonterminating errors
- not (logical NOT operator)
- notcontains (negated contains operator)
- notepad tool
- NoteProperty type
- notlike (negated like operator)
- notmatch (negated match operator)
- null character
- numbers 2nd 3rd
 - assigning to variables
 - bases for
 - numeric constants
- numeric format strings 2nd 3rd 4th





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

o format specifier

objects

- methods of

- output generating

- properties of

octal numbers

OdbcCommand class

OdbcConnection class

OdbcDataAdapter class

Operators

operators 2nd 3rd 4th 5th 6th

- arithmetic

- binary 2nd 3rd

- comparison 2nd

- logical

-or (logical OR operator)

OrderedDictionary class

Out verb

Out-String cmdlet

Outlook.Application object

OutlookExpress.MessageList object

output

output (command)

- capturing

- formatting 2nd 3rd 4th

-OutVariable parameter





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- P format specifier
- Page Down hotkey
- Page Up hotkey
- PasswordDeriveBytes class
- Path class
- Ping verb
- pipeline character (|)
- plus sign (+)
 - +?
- Pop verb
- popd command
- positional parameters
- PowerPoint.Application object
- PowerShell
 - .NET class representing
 - as interactive shell 2nd 3rd 4th 5th
 - documentation for
 - prompt for
 - technologies supported by 2nd 3rd
- PowerShell verbs 2nd 3rd 4th
- PowerShell window
- PowerShell.exe file
- precedence control ((...))
- Process class
- process keyword in scripts
- process statement
- profiles
- properties
 - instance
 - static
- PropertySet type
- providers
- PS > prompt
- PsBase
 - .psl file extension
- PSObject class 2nd
- PSObject type shortcut
- PSReference class
- Publish verb
- Publisher.Application object
- Push verb
- pushd command
- pwd command





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

quantifiers

question mark (?)

?!

?:

?<!

?<=

?=

?>

??

QuickEdit mode





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- R format specifier 2nd
- r option
- Random class
- RDS.DataSpace object
- Read verb
- Receive verb
- Ref type shortcut
- Regex class 2nd
- regex option
- Regex type shortcut
- registry
- Registry class
- RegistryKey class
- RegistrySecurity class
- regular expressions
 - backreference constructs
 - case sensitivity in
 - quantifiers
 - whitespace in
- Remove verb
- Rename verb
- replace operator
- Resolve verb
- Restore verb





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- s format specifier 2nd
- SAPI.SpVoice object
- scope of variables
- screen buffer
 - size
- script blocks
- ScriptBlock class
- ScriptBlock type shortcut
- Scripting.FileSystemObject object
- Scripting.Signer object
- Scriptlet.TypeLib object
- ScriptMethod type
- ScriptProperty type
- ScriptPW.Password object
- scripts
 - commands in
 - objects in
 - running
 - statement blocks in
 - writing
 - writing using history buffer
- Search verb
- SecureString class
- security
- Select verb
- Send verb
- SerialPort class
- Set verb
- Set-PsDebug cmdlet
- SHA1 class
- shortcuts for types
- Show verb
- single quote
- single quotes ('...')
- SmtplibClient class
- SoundPlayer class
- Split verb
- SqlCommand class
- SqlConnection class
- SqlDataAdapter class
- ss format specifier
- statement blocks 2nd
- static methods
- static properties
- Stop verbose preference
- Stopwatch class
- StreamReader class
- StreamWriter class
- String class
- string formatting
 - date and time 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
 - numeric 2nd 3rd
- StringBuilder class
- StringReader class
- strings 2nd 3rd

- escape sequences in
- replacing text in
- StringWriter class
- strongly typed arrays
- strongly typed variables
- substitution patterns
- subtraction operator (-)
- switch statement 2nd 3rd 4th
- Switch type shortcut
- SwitchParameter class
- System.Math class





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

T format specifier
t format specifier
tab
TcpClient class
Test verb
text format specifier
text selection
'text' format specifier
TextReader class
TextWriter class
Thread class
tokens
trace sources
Trace verb
Trace-Command cmdlet
tracing
trap statement
TripleDESCryptoService-Provider class
Truncate method
tt format specifier
Type class
type conversion operator (-as)
type operator (-is)
type shortcuts
TypeName class
TypeName type shortcut
types
types.ps1xml file





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- U format specifier
- Unix commands
- Unlock verb
- Unpublish verb
- Up arrow hotkey
- Update verb
- Update-FormatData cmdlet
- Update-TypeData cmdlet
- Uri class
- Use verb
- user interface





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Variable Provider
verbose output from commands
verbs 2nd 3rd
vertical tab





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

-w option

web site resources

- PowerShell documentation

- WMI documentation

WebClient class

WellKnownSidType class

WhatIf parameter

while statement

whitespace

-wildcard option

wildcards

- in Get-Command cmdlet

- in parameters

- in switch statement

Win32_BaseBoard class

Win32_BIOS class

Win32_BootConfiguration class

Win32_CacheMemory class

Win32_CDROMDrive class

Win32_ComputerSystem class

Win32_ComputerSystemProduct class

Win32_DCOMApplication class

Win32_Desktop class

Win32_DesktopMonitor class

Win32_DeviceMemoryAddress class

Win32_Directory class

Win32_DiskDrive class

Win32_DiskPartition class

Win32_DiskQuota class

Win32_DMACHannel class

Win32_Environment class

Win32_Group class

Win32_IDEController class

Win32_IRQResource class

Win32_LoadOrderGroup class

Win32_LogicalDisk class

Win32_LogicalMemoryConfigu-ration class

Win32_LogonSession class

Win32_NetworkAdapter class

Win32_NetworkAdapterConfigu-ration class

WIN32_NetworkClient class

Win32_NetworkConnection class

Win32_NetworkLoginProfile class

Win32_NetworkProtocol class

Win32_NTDomain class

Win32_NTEventlogFile class

Win32_NTLogEvent class

Win32_OnBoardDevice class

Win32_OperatingSystem class

Win32_OSRecoveryConfiguration class

Win32_PageFileSetting class

Win32_PageFileUsage class

Win32_PerfRawData_PerfNet_Server class

Win32_PhysicalMemoryArray class

Win32_PortConnector class

Win32_PortResource class
Win32_Printer class
Win32_PrinterConfiguration class
Win32_PrintJob class
Win32_Process class
Win32_Processor class
Win32_Product class
Win32_QuickFixEngineering class
Win32_QuotaSetting class
Win32_Registry class
Win32_ScheduledJob class
Win32_SCSIController class
Win32_Service class
Win32_Share class
Win32_SoftwareElement class
Win32_SoftwareFeature class
Win32_SoundDevice class
Win32_StartupCommand class
Win32_SystemAccount class
Win32_SystemDriver class
Win32_SystemEnclosure class
Win32_SystemSlot class
Win32_TapeDrive class
Win32_TemperatureProbe class
Win32_TimeZone class
Win32_UninterruptiblePower-Supply class
Win32_UserAccount class
Win32_VoltageProbe class
window
window size
Windows key + r hotkey
WindowsBuiltInRole class
WindowsIdentity class
WindowsPrincipal class
WMI (Windows Management Instrumentation)
 class categories
Wmi type shortcut
WmiClass type shortcut
WmiSearcher type shortcut
Write verb





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

- X format specifier
- XML as data type
- XML support
- Xml type shortcut
- XmlDocument class 2nd 3rd
- XmlElement class
- XmlTextWriter class
- xor (logical exclusive OR operator)





Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Y format specifier
y format specifier
yy format specifier
yyy format specifier
yyyy format specifier
yyyyy format specifier



Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

z format specifier
zz format specifier
zzz format specifier